VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

ZOBRAZENÍ 3D SCÉNY VE WEBOVÉM PROHLÍŽEČI

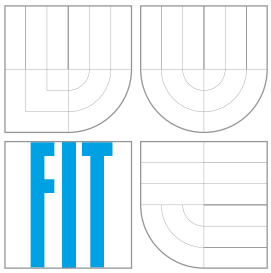
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. TOMÁŠ SYCHRA

BRNO 2013

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

ZOBRAZENÍ 3D SCÉNY VE WEBOVÉM PROHLÍŽEČI

DISPLAYING 3D GRAPHICS IN WEB BROWSER

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. TOMÁŠ SYCHRA

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. MICHAL ŠPANĚL, Ph.D.

BRNO 2013

Abstrakt

Tato práce zkoumá možnosti zobrazení akcelerované 3D scény v okně webového prohlížeče. Podrobněji se zabývá standardem WebGL a jeho využitím v praxi. V rámci práce byla vytvořena reálná aplikace pro vizualizaci volumetrických medicínských dat, která demonstruje současné možnosti webových technologií. Aplikace je založena na kombinaci JavaScriptu, WebGL a knihovny Three.js. Zobrazovaná data se načítají z externího úložiště Google Drive. Významnou část aplikace tvoří implementace 3D zobrazení volumetrických dat (tzv. volume rendering) s využitím metody Ray-casting a standardu WebGL.

Abstract

This thesis discusses possibilities of accelerated 3D scene displaying in a Web browser. In more detail, it deals with WebGL standard and its use in real applications. An application for visualization of volumetric medical data based on JavaScript, WebGL and Three.js library was designed and implemented. Image data are loaded from Google Drive cloud storage. An important part of the application is 3D visualization of the volumetric data based on volume rendering technique called Ray-casting.

Klíčová slova

3D grafika, webový prohlížeč, akcelerovaná grafika, WebGL, medicínská data, objemová data, volume rendering, Volume ray casting, Marching cube, frameworky, vizualizace dat

Keywords

3D graphics, web browser, accelerated graphics, WebGL, medical data, volumetric data, volume rendering, Volume ray casting, Marching cube, frameworks, data visualization

Citace

Tomáš Sychra: Zobrazení 3D scény ve webovém prohlížeči, diplomová práce, Brno, FIT VUT v Brně, 2013

Zobrazení 3D scény ve webovém prohlížeči

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Michala Španěla, Ph.D.

.....
Tomáš Sychra
21. května 2013

Poděkování

Tímto bych chtěl poděkovat vedoucímu své diplomové práce, Ing. Michalu Španělovi Ph.D., za pomoc a čas, který mi věnoval. V neposlední řadě bych rád poděkoval i svým rodičům a přítelkyni za jejich velkou podporu.

Obsah

1 Úvod **3**

2 Zobrazení akcelerované 3D scény v okně webového prohlížeče **4**
2.1 Historie 4
2.2 Současný trend 4

3 Technologie WebGL **6**
3.1 Charakteristika 6
3.2 Bezpečnost 7
3.3 Podpora 7
3.4 Knihovny 8
3.5 Budoucnost 9

4 Vizualizace objemových dat **10**
4.1 Použití v praxi 10
4.2 Zdrojová data 11
4.3 Osvětlovací model 12
4.4 Nepřímé renderovací techniky 12
4.5 Přímé renderovací techniky 14

5 Vizualizace objemových dat ve WebGL **17**
5.1 Existující aplikace 17
5.2 Známé problémy 17
5.3 Řešení pomocí algoritmu Volume ray casting 18
5.4 Možné optimalizace 19

6 Návrh programu pro vizualizaci medicínských dat **20**
6.1 Základní specifikace aplikace 20
6.2 Návrh rozhraní 20
6.3 Uživatelské účty 22
6.4 Datová úložiště 22
6.5 Multiplanární zobrazení 23
6.6 Zobrazení jednotlivých řezů 24
6.7 Volume rendering 24

7 Implementace aplikace **31**
7.1 Použité technologie 31
7.2 Inicializace aplikace 32

7.3 Běh programu 32
7.4 Architektura aplikace 32
7.5 Uživatelské účty 34
7.6 Příprava dat 35
7.7 Volume rendering 36

8 Vyhodnocení dosažených výsledků **40**

9 Závěr **44**

A Obsah CD **47**

B Plakát **48**

Kapitola 1

Úvod

V současné době dochází k mohutnému stěhování aplikací na internet a do cloudu. Chceme-li využít plný potenciál internetu, který je nám nabízen, nemůžeme se omezit na jednoduché aplikace, jenž nám zprostředkovávají náhled na data pouze ve dvou dimenzích.

Vysoká rychlost internetu je současně se vzrůstajícím výkonem osobních zařízení a optimalizací interpretace JavaScriptu nezbytnou podmínkou pro vznik komplexních aplikací běžících nativně ve webovém prohlížeči. Abychom byli schopni takto složité aplikace tvořit, je třeba mít k dispozici hardwarovou akceleraci 3D grafiky.

Tématem mé diplomové práce je zobrazení 3D scény v okně webového prohlížeče a prozkoumání nejnovějších poznatků z tohoto odvětví. Proto jsem si zvolil za cíl vytvoření komplexního software pro vizualizaci medicínských dat, který bude fungovat nativně v prohlížeči s využitím stále poměrně nové technologie WebGL a nad ní postavenou knihovnou Three.js. Hlavním implementačním jazykem je JavaScript. Tuto kombinaci technologií považuji za nejvýhodnější variantu. Software demonstruje možnosti, které jsou v současné době dostupné.

Práce navazuje na mou bakalářskou práci, v rámci níž jsem rovněž implementoval aplikaci zobrazující medicínská data. Byla ale velmi jednoduchá. Podporovala pouze multiplanární zobrazení. Založena byla na frameworku O3D, který bohužel zanikl. Celou aplikaci tedy bylo nutné od základů přepsat. Využil jsem předchozích zkušeností z bakalářské práce a změnil jsem původní návrh. Mezi nové funkce patří vykreslování jednotlivých řezů (axiální, sagitální a koronární řez). Klíčová je podpora volumetrického zobrazení (módy X-Ray, MIP, Shaded), propojení se vzdálenými úložišti (Google Drive) a možnost načítat data i z lokálních úložišť.

V první kapitole je rozebráno, jak probíhal vývoj, který zapříčinil vznik standardu WebGL, a kterým směrem se bude oblast pravděpodobně vyvíjet. Druhá kapitola se již zaobírá konkrétně technologií WebGL. Podrobně popisuje klíčové vlastnosti, které standard definuje. Dále se věnuje bezpečnosti a popisuje největší známá bezpečnostní rizika. Podstatnou částí této kapitoly je také přehled aktuálních knihoven, které práci s WebGL velmi ulehčují a značně zrychlují. Třetí kapitola se věnuje vizualizaci objemových dat v obecné rovině (nezávisle na standardu, který je použit pro implementaci). Čtvrtá kapitola se konkrétně zaměřuje na možnosti algoritmů pro vizualizace objemových dat v kombinaci s WebGL. V páté kapitole se věnuji návrhu cílového programu a technologiím, které jsem pro něj zvolil. Detailně rozepisuji očekávané vlastnosti aplikace i to, jak některých z nich teoreticky dosáhnu. Kapitola věnující se implementaci obsahuje základní popis objektů, ze kterých je program složen. Velmi podrobně se věnuje implementaci volumetrického zobrazení, kde jsou nastíněny i klíčové části fragment shaderu.

Kapitola 2

Zobrazení akcelerované 3D scény v okně webového prohlížeče

Tato kapitola je věnována historii 3D zobrazení ve webových prohlížečích. Dále pojednává o možném budoucím vývoji této technologie.

2.1 Historie

Některé firmy si uvědomily, že by bylo vhodné vytvořit technologii, která by umožňovala 3D grafiku v okně webového prohlížeče akcelerovat přímo na grafické kartě. Do této doby bylo nutné všechny výpočty ponechat pouze na procesoru, který s nimi měl velké problémy, a bylo tedy možné zobrazovat pouze velmi jednoduché věci (procesor simuloval grafické výpočty). To však bylo pro rozsáhlejší aplikace neúnosné a zcela nepoužitelné. Co tedy s tím? Začaly vznikat různé doplňky do prohlížečů.

První přišel na scénu Java applet (JOGL - Java OpenGL)[1], který vyvíjela skupina Sun Microsystems Game Technology Group. Applet zpřístupňoval většinu funkcí ze standardu OpenGL. Následně se objevil Stage3D [2] od firmy Adobe. Stage3D API a Adobe AIR nabízí plnou hardwarovou akceleraci jak ve všech desktopových prohlížečích, tak i na mobilních platformách. Jedná se o velmi kvalitní a výkonný plugin. Bohužel stále se jedná a plugin a s tím se někteří velcí hráči na trhu nehodlali smířit (například Google). Google sám ze začátku zaštiťoval projekt O3D (což byl také plugin), které umožňoval hardwarovou akceleraci na grafické kartě, ale později od něj upustil.

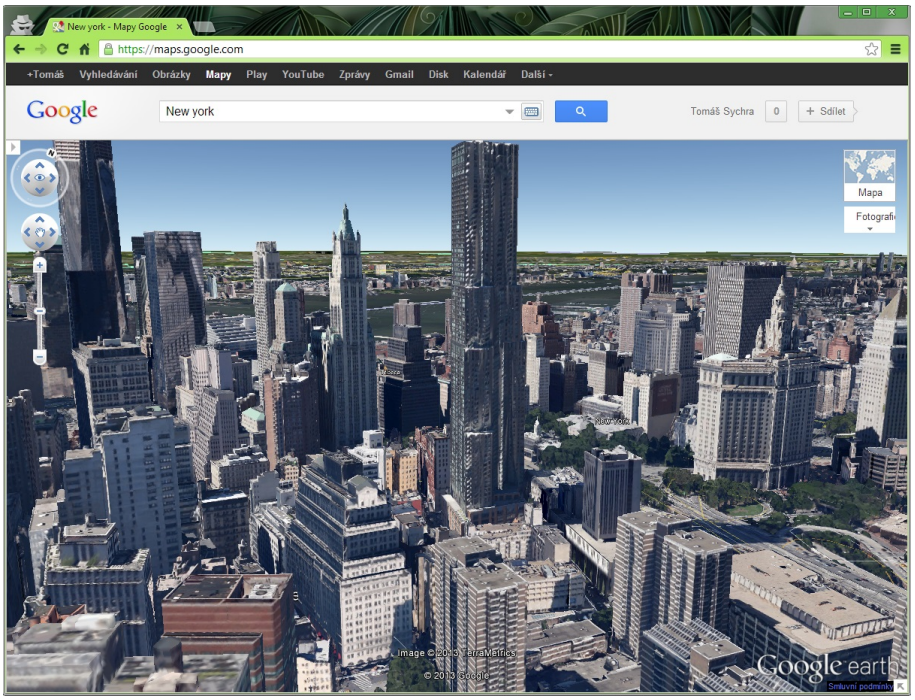
Bylo by vhodné zde zmínit ještě jednu technologii. Jedna z možností jak akcelerovat 3D grafiku v prohlížeči je použití platformy Silverlight 3D [3] od Microsoftu. Ta je určena pro vývoj pokročilých aplikací, jenž mají běžet ve všech běžně používaných prohlížečích (Google Chrome, Internet Explorer, Mozilla Firefox a Safari) nebo v režimu „Out of browser“ [4], tedy mimo prohlížeč ve vlastním okně. První verze byla vydána již v roce 2007, ale ještě nepodporovala akceleraci 3D grafiky. Jako první HW akceleraci 3D grafiky podporovala až třetí verze, která byla vydána v roce 2009. Technologie je to jistě dobrá, ale opět se jedná o doplněk pro prohlížeč.

2.2 Současný trend

V dnešní době je na scéně zdánlivě jednoduchý trend. Vše by mělo být velmi snadné. Za ideální přístup se považuje takzvané „Ovládání bez návodu“. Tato ideologie je v silném roz-

poru s přístupem doinstalování jakéhokoli přídavného doplňku pro prohlížeč. Cílem je, aby všechny technologie byly integrovány, a uživatel jen zadal URL adresu programu. Všechno musí fungovat tak jak má, bez jakéhokoli zásahu ze strany uživatele. K tomu bylo třeba vyvinout zcela nový standard WebGL [5]. WebGL je technologie, jenž je dnes přímo integrována ve většině webových prohlížečů. Výjimkou je Internet Explorer, jelikož Microsoft má připomínky k bezpečnosti tohoto standardu a odmítá ho integrovat. Bezpečnostní rizika jsou pro něj příliš velká.

Dále je v kurzu stěhování všemožných dat do vzdálených úložišť (cloudu). Jedná se o velmi praktickou záležitost, jelikož každý ocení mít svá data všude a vždy dostupná, kdykoli si na ně vzpomene. Tento trend velmi přichází vhod i webovým aplikacím, které mohou uživatelská data na vybraná úložiště ukládat. Vzniká tak ideální kombinace. Uživatel může webovou aplikaci pustit kdykoli, odkudkoli a současně v ní má uložena vždy aktuální data, která jsou navíc automaticky zálohována. Příkladem mohou být mapy[1], které umožňují zobrazit vymodelované budovy ve 3D.



Obrázek 2.1: Google Mapy - Zobrazení 3D budov pomocí WebGL.

[1] http://maps.google.com

Kapitola 3

Technologie WebGL

Kapitola shrnuje podstatné informace standardu jako je například jeho podpora ve webových prohlížečích či nejlepší frameworky a knihovny. Upozorňuje také na možná rizika spojená s webovými aplikacemi v kombinaci se standardem WebGL.

3.1 Charakteristika

Původním autorem standardu WebGL byla Mozilla Foundation, později však vývoj přešel pod skupinu Khronos Group, která ho vyvíjí dodnes. První oficiální specifikace vyšla v březnu roku 2011.

Standard WebGL vychází z již existujícího standardu OpenGL ES 2.0, což sebou nese jisté výhody i nevýhody. OpenGL ES (Embedded System) je určen pro mobilní zařízení. WebGL má tak zaručenu jistou přenositelnost i na mobilní zařízení. Značnou výhodou také je, že postupy, na které jsou programátoři zvyklí z OpenGL, jsou ve WebGL víceméně stejné. Nemusí se tudíž nic nového učit. Problém naopak nastává právě u vizualizace medicínských dat, protože standard OpenGL ES 2.0 nemá podporu pro objemové textury. Je tedy problém s vizualizací volume renderingu.

WebGL je plně integrováno s rozhraním DOM (Document Object Model). Může být používáno jakýmkoli DOM kompatibilním jazykem (Java, JavaScript).

V praxi se však používá pouze v kombinaci s JavaScriptem. Ten se stará o inicializaci WebGL a také o přípravu dat, které bude WebGL zobrazovat. Jelikož je JavasSript interpretovaný jazyk, nepatří mezi nejrychlejší, ačkoli v posledních letech prožil mohutný vývoj a jeho rychlost se velmi zvýšila. Za tímto pokrokem z velké části stojí společnost Google, která do webových aplikací vkládá velkou naději a mnoho sil. Výhodou JavaScriptu je jeho pružnost (někdy je to až na škodu) a tudíž velmi lehká rozšiřitelnost. Bylo nutné ho rozšířit o typovaná pole, jelikož bez nich se WebGL neobejde.

Stejně jako aktuální verze ostatních GL standardů, tak i WebGL podporuje programovatelnou pipeline využívající shadery psané v jazyce GLSL (OpenGL Shading Language). GLSL je vyšší programovací jazyk pro psaní shaderů, založený na syntaxi jazyka C. Kód shaderů se většinou vkládá přímo do zdrojového kódu webové stránky. Lze je však ukládat i do souborů a při načítání stránky je dynamicky nahrávat.

WebGL je nízkoúrovňový programovací jazyk, tudíž je v základní podobě (stejně jako OpenGL) relativně složité. Většinu těchto nedostatků řeší knihovny. Občas je však vhodné mít kód pevně pod kontrolou, a to platí i v tomto případě.

Vytvoření jednoduché aplikace není nijak složité. Základem je prvek `<canvas>` ze standardu HTML5. Pak už vše funguje jako v OpenGL. [5]

3.2 Bezpečnost

Bezpečnostní hrozby existují a jejich zneužití je bezesporu reálně aplikovatelné. Příčinou ale není samotné WebGL, ale spíše software třetích stran (ovladače grafických karet), ve kterém jsou velmi často závažné chyby a nikdo jiný než vývojáři grafických ovladačů tyto chyby neopraví. V minulých dobách tyto chyby nebyly zásadním problémem, protože je nebylo tak jednoduché zneužít. WebGL ale zpřístupňuje přímý přístup k těmto ovladačům útočníkům z webu a právě v tomto místě nastává závažný problém. Z těchto důvodů odmítá Microsoft integrovat WebGL do Internet Exploreru. [6]

Nejzásadnější problémy [6]

- WebGL zpřístupňuje webové stránce téměř neomezený přístup ke grafickému hardware. Lze tak lehce využít chyb v ovladačích.
- Zneužitím kódu shaderů nebo dodáním extrémního množství geometrie lze provést DoS útoky.
- Bezpečnost příliš závisí na grafických ovladačích (software třetích stran), tudíž vývojáři aplikací nejsou schopni zaručit, že vše bude zabezpečené. Nikdo totiž nezaručí, že bude výrobce dostatečně rychle reagovat na nově nalezené chyby. Stejně tak nikdo nezaručí, že bude uživatel udržovat ovladače aktuální.

Důležitá bezpečnostní omezení ukládaná specifikací WebGL[5]

- Omezení zdrojů - Zdroje, se kterými WebGL pracuje, mezi které patří textury a VBOs, musí být vždy inicializovány, a to i v případě, že byly vytvořeny, aniž by je uživatel inicializoval nějakými počátečními daty (což se běžně dělá k rezervaci místa pro textury nebo VBO, když ještě data nemáme). Pokud tato inicializační data nejsou poskytnuta, musí WebGL samo inicializovat buffery na 0.
- Omezení původu dat - Aby WebGL zamezilo úniku informací, zakazuje nahrávat jako textury elementy videa nebo obrázku, které nemají stejný původ jako je původ dokumentu, jenž obsahuje prvek canvas s WebGL contextem, a zároveň vytvářet textury z elementů, které nemají *origin-clean* flag nastaven na false.

3.3 Podpora

Jak je uvedeno v [7], všechny hlavní prohlížeče, s výjimkou Internet Exploreru, WebGL podporují. Začíná se prosazovat i podpora v mobilních zařízeních (zatím převážně na Androidu).

Podpora na desktopu

- Google Chrome / Chromium - WebGL je dostupné ve stabilní verzi Chrome (od verze 9) i ve stabilní verzi Chromium. Na mobilních platformách zatím nativně nefunguje (funguje pouze při manuálním zapnutí).

- Mozilla Firefox - podporuje WebGL od verze 4.0
- Safari - ve verzi 5.x nebo vyšší
- Opera - podpora je zahrnuta ve verzi 12 Beta, ale musí se explicitně zapnout
- Internet Explorer - standard WebGL není podporován a v blízké době nebude; existuje možnost instalace pluginu, který běh WebGL aplikací umožňuje [8]

Mobilní zařízení

- Google Chrome
- Firefox for mobile - podpora pouze na Androidu a to pouze v nestabilní verzi od roku 2011
- Opera mobile - ve verzi 12 (pouze na Androidu)
- Tizen 1.0

3.4 Knihovny

WebGL je nízkoúrovňový jazyk a napsat v něm i jednoduchý program trvá poměrně dlouho. Ze začátku mnoho lidí namítalo, že je to velký nedostatek a odmítalo WebGL používat. Naštěstí již vzniklo mnoho frameworků a knihoven. V dnešní době není nutno vše ručně psát a práce se tak extrémně zrychlila. Nicméně si myslím, že je velmi důležité si pár jednodušších programů napsat, protože člověk pak má přesnější představu o tom, co program ve skutečnosti dělá. Mezi nejznámější knihovny [9] patří C3DL, CopperLicht, GLGE, SceneJS, Three.JS, X3DOM a OSG.JS. Některé z nich budou popsány níže. Další k nalezení zde[1].

C3DL [10] (Canvas 3D JS Library) je JavaScriptová knihovna, která velmi ulehčuje psaní 3D aplikací využívajících WebGL. Poskytuje nástroje (matematické funkce, správu scény, třídy na 3D objekty a další). Umožňuje tak programovat pokročilé 3D aplikace, aniž by programátor musel znát hlouběji WebGL. Dokumentace je zpracována dobře. Existuje také velké množství tutoriálů[2]. Není tedy problém s touto knihovnou pracovat během krátkého času značně efektivně.

CopperLicht [11] je komerční 3D engine. Jedná se o čistou implementaci v JavaSriptu, aniž by používal jakýkoli plugin. Jeho hlavní výhodou je editor CopperCube, který umožňuje vytvářet například hry i uživatelům, kteří WebGL a této problematice příliš nerozumí. Bohužel zdarma je pouze pro nekomerční použití. K dispozici jsou i tutoriály[3].

SceneJS (3D scene Graph Engine for WebGL)[12] je jednoduchý open-sourcový 3D engine. Graf scény má založený na JSONu. Jeho klíčové vlastnosti lze nalézt na jeho wiki v sekci Features[4]. Jedna z nejdůležitějších předností je JSONová reprezentace scény (Logical JSON API). Data se lehce exportují, ukládají do databáze, ale jsou i lidsky čitelná.

[1] http://www.khronos.org/webgl/wiki/User_Contributions
[2] http://www.c3dl.org/index.php/tutorials/
[3] http://www.ambiera.com/copperlicht/tutorials.html
[4] https://github.com/xeolabs/scenejs/wiki/Features

Three.JS [13] je knihovna vydaná pod MIT licencí, což je zcela ideální i pro komerční vývoj. Jedná se o nenápadnou knihovnu, která však vyniká svými vlastnostmi, malou velikostí a jednoduchou implementací. Řízení celého programu ponechává zcela na programátorovi, což je značná výhoda. Ten tak neztrácí přehled a vše má pevně v rukou (na rozdíl od většiny ostatních frameworků). Klíčové vlastnosti jsou uvedeny na wiki[5]. Umí renderovat do textur, obsahuje picking manager, materiály, animace a mnoho dalších předpřipravených struktur. Nevýhodou je nekompletní dokumentace [6], kde se na mnoha místech stále vyskytuje značka TODO (což nevypadá příliš věrohodně). Nekompletní dokumentaci nahradí velké množství dobrých příkladů [7].

OSG.JS [14] je JavaScriptovou implementací OpenSceneGraph (OSGJS je založen na konceptu OpenSceneGraph). Stále je aktivně vyvíjen, ale dokumentace je špatná. Modely jsou uloženy a organizovány jako graf. Podporuje materiály, textury, shadery, osvětlení, kameru, transformace a animace. Jakákoli fyzika ale chybí (interakce objektů). Chybí i třídy pro pokročilé animace. Organizace tříd by mohla být lepší.

Na závěr bych rád zmínil jeden nástroj, který velmi pomáhá při vývoji. Jedná se o WebGL Inspector [8], jenž umožňuje přímo sledovat data na grafické kartě. Při chybě v shaderu přesně ukáže a označí místo, kde došlo při kompilaci programu k chybě. Značně usnadňuje práci.

Pro běžné ladění JavaScriptu zcela postačí integrované konzole v prohlížečích. Ty jsou dnes na velmi vysoké úrovni (umožňují například prohlížení paměti, krokování skriptu).

3.5 Budoucnost

Když přišlo WebGL na scénu, jeho budoucnost byla značně nejistá. V poslední době to ale vypadá velmi optimisticky. Jak bylo rozebráno výše, až na IE všechny prohlížeče nativně standard WebGL podporují. Rychlost JavaScriptu se v poslední době velmi zvýšila. Začínají vznikat i reálné aplikace, které jsou opravdu užitečné. WebGL velmi nahrává i skutečnost, že za jeho vývojem stojí velcí hráči jako je Mozilla Foundation a Google. Vývoj by snad mohla zpomalit jen bezpečnostní rizika, která se ale postupně daří eliminovat (např. použitím „sandboxingu“). V neposlední řadě nahrává tomuto standardu i nálada současné doby, kdy opět dochází k centralizaci aplikací na web. Je „in“ mít všechna data na vzdálených úložištích a neustále dostupná odkudkoli a kdykoli. V kombinaci s webovými aplikacemi se utváří ideální produkt.

[5] https://github.com/mrdoob/three.js/wiki/Features
[6] http://mrdoob.github.com/three.js/docs/54/
[7] http://stemkoski.github.com/Three.js/
[8] http://benvanik.github.com/WebGL-Inspector/

Kapitola 4

Vizualizace objemových dat

Kapitola se věnuje přímým a nepřímým vykreslovacím technikám. Pojednává také o možném využití prezentovaných algoritmů v praxi.

4.1 Použití v praxi

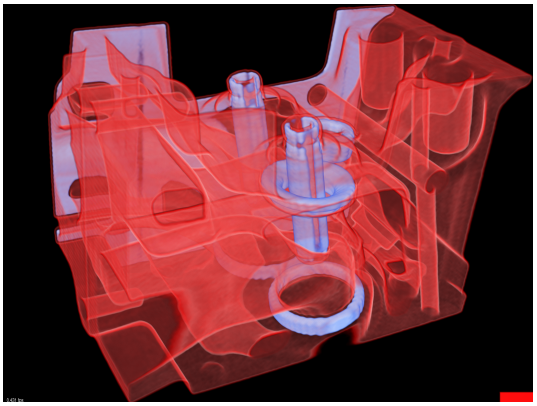
Vizualizace objemových dat je důležitá. Bohužel není jednoduché vizualizovat 3D data v reálném světě (fyzicky). Existují prototypy 3D displejů, ale jejich masové rozšíření se nekoná a asi ještě nějakou dobu konat nebude. Proto je nutné vyřešit problém, jak efektivně zobrazovat objemová data na 2D displejích. Řešením je technika zvaná volume rendering. Jedná se o 2D projekci 3D dat tak, aby byl uživatel schopen lehce pochopit jejich strukturu (přímo na 2D se obraz převádí pouze u přímých renderovacích technik). Bez této techniky by si lidé některá data dokázali jen těžko představit. Jako nevýhoda se může zdát zdánlivý zánik detailů v datech, ale to je nutné a žádoucí, abychom byli schopni pochopit data komplexně (jistá abstrakce je důležitá). Pro detailní zkoumání oblastí snímků je vhodnější jiný typ vizualizace. Například multiplanární zobrazení objemových dat.

V praxi se používá všude, kde je potřeba interpretovat objem nějakého tělesa. Tento dokument se bude zabývat pouze volumetrickým zobrazením medicínských dat (4.2), ale stejně tak se s tímto zobrazením v praxi můžeme setkat i ve strojírenství (4.1) a jiných oborech (např. meteorologie).

Existuje mnoho softwarů pro vizualizaci volumetrických dat. Bohužel programy, jenž lze opravdu nazvat použitelnými a komplexními, běží klasicky pouze offline. Jak bylo psáno v úvodu, cílem práce je stvořit reálnou aplikaci, která poběží online a je na konkrétním počítači zcela nezávislá. Následující software se stal inspirací při návrhu výsledného programu.

- Slicer3D [1] - jedná se o open-source (BSD licence); multiplatformní
- OsiriX Imaging software [2] - není multiplatformní (pouze pro Mac OS X)
- ImageVis3D [3]
- 3DimViewer [4]

[1] http://www.slicer.org/pages/Introduction
[2] http://www.osirix-viewer.com/
[3] http://www.sci.utah.edu/cibc-software/imagevis3d.html
[4] http://www.3dim-laboratory.cz/cz/software/3dimviewer/

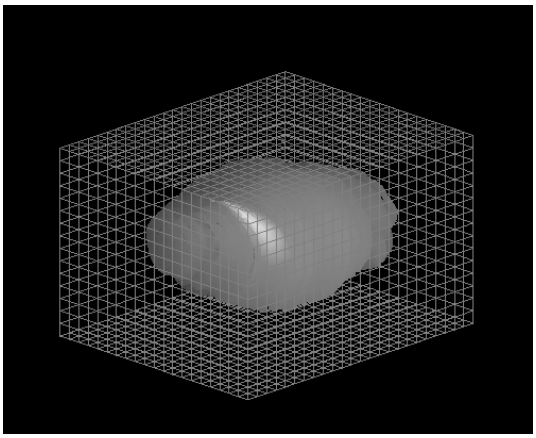


Obrázek 4.1: Motor [15].

Obrázek 4.2: Příklad zobrazení CT dat lebky pomocí volume renderingu [16].

4.2 Zdrojová data

Zdrojová data jsou nejčastěji sbírána pomocí CT (Computed Tomography), kdy je objekt nasnímán řez po řezu. Jednotlivé řezy jsou složeny do kvádru, jak je názorně vidět na obr. (4.3). Povšimněme si, jak je velký kvádr dělen na malé kvádry. Ty jsou odborně nazývány voxely. V programování se zdrojová data reprezentují polem o třech rozměrech (3D pole - 3D mřížka). Z těchto dat jsme schopni segmentováním, nastavováním průhlednosti, nastavováním prahů, barev a dalších vlastností zjistit extrémně velké množství informací. Nesou kompletní informaci o vnitřní struktuře a povrchu objektu. Naopak nenesou informaci o barvě (tu je nutno ručně přiřadit).



Obrázek 4.3: Organizace objemových dat [17].

4.3 Osvětlovací model

Osvětlení je důležitá součást realistického vykreslení objektu. Umožňuje zvýraznit i drobné nerovnosti. V následujícím odstavci je vysvětlen nejzákladnější typ osvětlení.

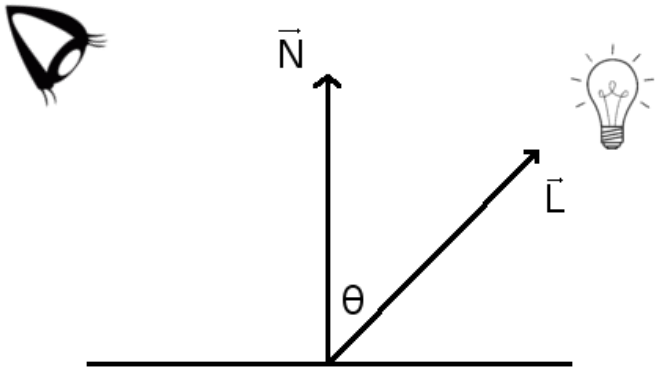
Lambertův osvětlovací model se počítá dle vzorce 4.3. Ten popisuje jak povrch voxelu odráží světlo vycházející ze světelného zdroje. Princip je znázorněn na obrázku č. 4.4. N označuje normálu povrchu voxelu (gradient) a L zde označuje vektor směřující od voxelu do světla. Oba vektory musí být normalizované, jinak by nebylo možné použít zjednodušený výpočet dle 4.3, ale bylo by nutné počítat náročnější výpočet dle 4.1 a 4.2.

$$I = I_p k_d cos(\theta) \tag{4.1}$$

$$v_1 * v_2 = |v_1||v_2|cos(\theta) \tag{4.2}$$

$$I = I_p k_d (N * L) \tag{4.3}$$

kde I_p je intensita bodového světla, k_d jsou difúzní vlastnosti povrchu voxelu a θ je úhel svíraný normálou voxelu a vektorem směřujícím ke světlu.



Obrázek 4.4: Lambertův osvětlovací model.

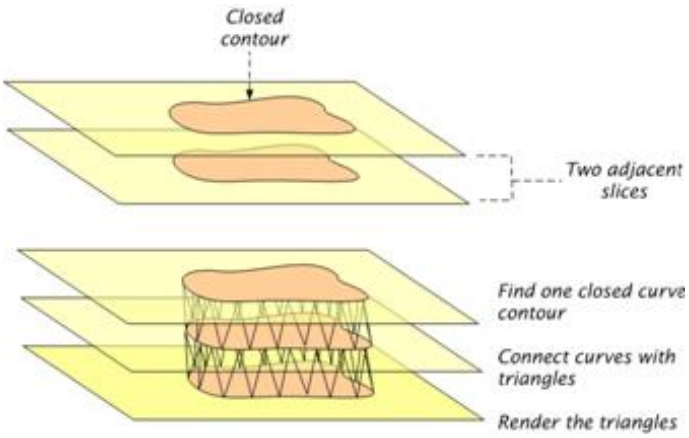
4.4 Nepřímé renderovací techniky

Vizualizace objemových dat může být provedena mnoha způsoby. Dělí se na dvě větší skupiny. Způsoby dělíme na techniky přímé a nepřímé. V této kapitole, jak již sám název napovídá, rozeberu dvě významné skupiny nepřímých renderovacích technik volumetrických dat.

Nepřímé renderovací techniky (povrchově orientované) označují takovou skupinu technik, jenž se nesnaží přímo vizualizovat jednotlivé voxely, ale nejdříve napasují (vygenerují) novou geometrii na volumetrická data, a teprve po vygenerování geometrie dle struktury dat tuto geometrii vykreslí.

Samotné generování geometrie dle objemových dat je poměrně náročnou záležitostí. Většinou se spojují oblasti, ve kterých mají voxely přibližně stejnou hodnotu (určenou uživatelem). Výsledek bývá reprezentován množinou polygonů představujících původní volumetrická data.

Surface tracking [18] je proces, při kterém dochází k rekonstrukci povrchu objektu nalezením kontur v každém řezu a jejich následným propojením pomocí geometrie (nejčastěji trojúhelníků). Kontury v řezech se hledají například prahováním. Jak technika funguje je názorně vidět na obr. 4.5.



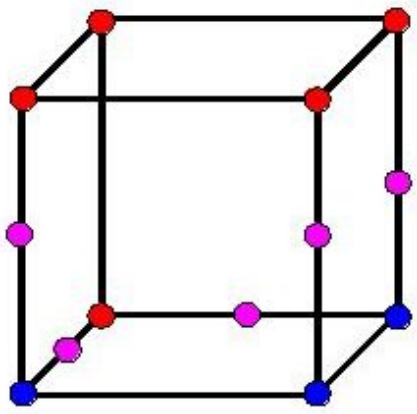
Obrázek 4.5: Technika Surface tracking [18].

Iso-surfacing [19, 18] je technika spojování voxelů po povrchu objektu, které mají stejnou hodnotu. Hodnota je předem definována uživatelem. Tato technika má mnoho výhod, ale i nevýhod. Jednou z výhod je, že výsledný model je opravdovým 3D modelem (může sloužit jako vstup do 3D tiskáren a jiných zařízení, kde je potřeba znát 3D reprezentaci objektu). Občas může pro uživatele být problém interpretovat získaná data (model), popřípadě může být vygenerovaný model značně zavádějící. Dále pak dochází ke ztrátě informace uvnitř objektu (protože získáváme jen povrchovou reprezentaci). S obarvením objektu podle intensity je samozřejmě problém také. Může být velmi složité model nějak smysluplně obarvit na základě intensity. Stává se, že je pak model nesrozumitelný a obarvení nedává příliš smysl. Většina modelů je proto generována monochromaticky. Problém taky nastává při samotném převodu objemových dat na polygonální model. Je-li v datech příliš mnoho výrazného šumu, vzniknou nám v cílovém modelu nežádoucí artefakty, které je třeba dodatečně vyhlazovat. Jedním z nejznámějších algoritmů pro hledání povrchu objektu ve volumetrických datech je Marching cubes.

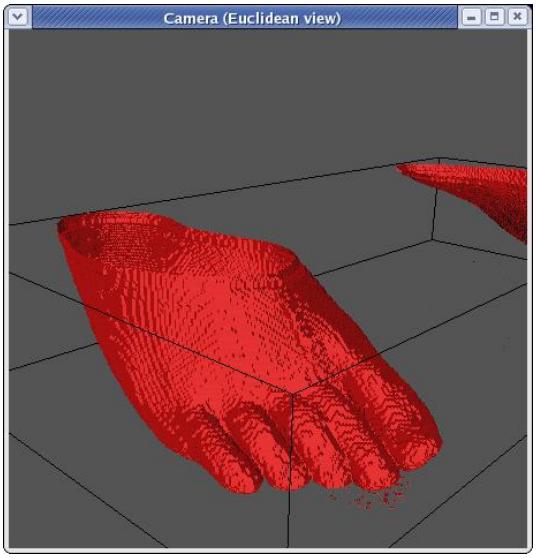
Algoritmus Marching cubes [20] spadá do skupiny iso-surface algoritmů. Funguje na poměrně jednoduchém principu. Objemový prostor rozdělí na jednotlivé kostky. Jejich vrcholy odpovídají jednotlivým voxelům v řezech (jedna kostka je tvořena z voxelů nacházejících se ve 2 řezech) vedle sebe. Každý vrchol kostky se pak porovnává s prahem, který reprezentuje hodnotu intenzity vrcholu objektu. V závislosti na výsledku porovnání se vrchol vyhodnocuje následovně. V případě, že vrchol spadá do výsledného objektu (hodnota <= práh), pak se ohodnocuje jako vnitřní. V opačném případě jako vnější. Ilustrace je znázorněna na obrázku č. 4.6.

Červené body na tomto obrázku do výsledného modelu spadají, zatímco modré nikoli. Je tedy zřejmé, že výsledný povrch bude s největší pravděpodobností mezi těmito vrcholy (mezi modrými a červenými - růžové body). Mezi růžovými body se následně vytváří plochy, které reprezentují hranici objektu. Je samozřejmě zapotřebí jistý post-processing na vyhlazení

povrchu. Povrch vygenerovaný algoritmem Marching cubes bez post-processingu (je na něm názorně vidět jak algoritmus funguje) je na obr. 4.7.



Obrázek 4.6: Jedna kostka v algoritmu Marching cubes [21].



Obrázek 4.7: Výsledek algoritmu Marching cubes [21].

4.5 Přímé renderovací techniky

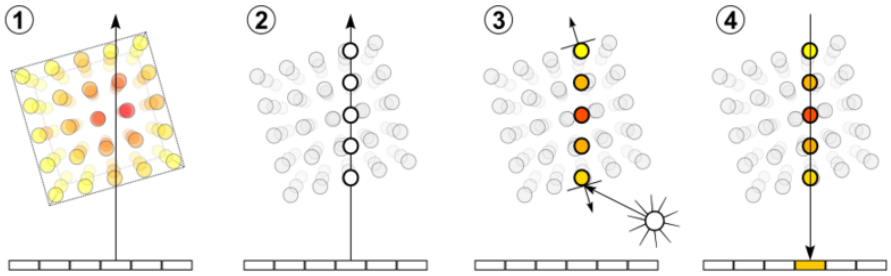
Přímé renderovací techniky umožňují vizualizovat objemová data přímo formou 2D obrazu. K objemovým datům přistupují zcela odlišně než nepřímé metody. Snaží se vytvořit cílový obraz tak, aby každý voxel byl potenciálně viditelný (snaží se využít všechny informace a při zobrazení žádné nezanedbat). Algoritmus přiřazuje každému voxelu barvu a průhlednost a následně utváří výsledný obraz. Barvu a průhlednost přiřazuje voxelu převodní funkce.

Níže je popsána základní teorie algoritmů využívaných k přímému zobrazení objemových dat.

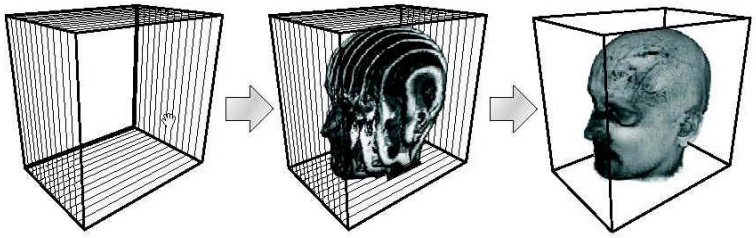
Volume ray casting [22, 18] je „image order“ algoritmus, což znamená, že jsou postupně počítány hodnoty výsledného obrazu (pixel po pixelu). Princip je následovný (lze sledovat na obr. 4.8). Z kamery jsou vyslány paprsky do scény přes zobrazovací rovinu (skrz každý pixel). Je potřeba sledovat pohyb každého paprsku při průchodu objektem, následně sečíst hodnoty jednotlivých voxelů (popřípadě násobit) a aplikovat na ně mapovací funkci. Výsledná hodnota se přiřadí příslušnému pixelu, kterým byl paprsek vyslán.

Texture mapping [22, 18] spadá do kategorie „Texture-based“ (na texturách založených) metod. Hlavní výhoda tohoto algoritmu je, že lze použít i tam, kde není implementována HW podpora 3D textur. Nemáme-li podporu 3D textur, musíme v paměti udržovat tři sady plátů (ty reprezentují objemová data - obr. 4.10), přičemž každá sada je kolmá k jedné ze tří os. Při změně kamery o více než 90° se mění aktuálně zobrazovaná sada. Tím je zaručena maximální možná odchylka řezů 45° od směru pohledu. Výsledek je vidět na obr. 4.9. Kvalita bez dostupnosti 3D HW podpory není tak velká. Máme-li podporu 3D

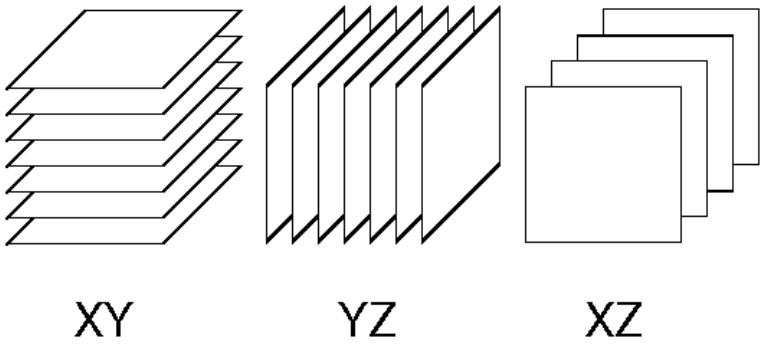
textur, pak nepotřebujeme 3 sady plátů, ale jen jednu, kterou orientujeme kolmo na pohled uživatele. Tím zaručíme nejlepší kvalitu.



Obrázek 4.8: Základní kroky Volume ray castingu [23].



Obrázek 4.9: Konečné zobrazení [24].



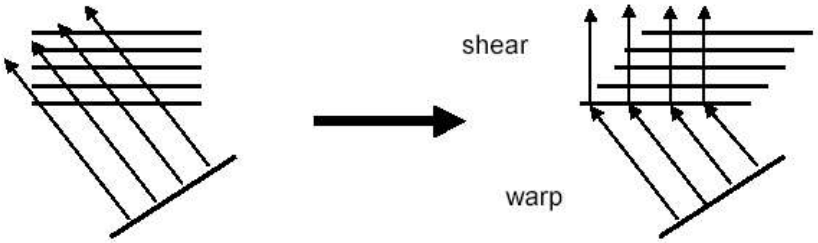
Obrázek 4.10: Osově zarovnané řezy [24].

Splatting [22, 18] nemá tak kvalitní výsledky jako ray casting, ovšem je rychlejší. Rozděluje těleso na voxely a ty pak promítne na projekční plochu do kruhu (jako když se rozpleskne sněhová koule). Takto se na projekční plochu postupně promítnou voxely ze všech plátů. Výhodou je, že výsledná interpolace probíhá až po promítnutí ve 2D (což je značně rychlejší).

Shear Warp [22, 18] je poměrně nová metoda, která je založena na metodě Texture mappingu bez použití 3D textur. Snaží se však odstranit jeho nedostatky (artefakty vznikající při špatných úhlech natočení kamery). Náročnost na paměť je stále velká, jelikož je

třeba, stejně jako u Texture mappingu, držet v paměti více setů dat. Hlavní myšlenka je znázorněna na obr. 4.11. Jednoduše řečeno funguje následovně.

- Proveď shear (posunutí řezů)
- Projekce řezů a získání obrazu
- Pozměň obraz (warp) tak, aby odpovídal natočení kamery



Obrázek 4.11: Princip Shear Warp algoritmu [22].

Kapitola 5

Vizualizace objemových dat ve WebGL

5.1 Existující aplikace

- X Toolkit for Scientific Visualization [25] je zaštiťovaný Children's Hospital of Boston a Harvard Medical School. Je vydán pod licencí MIT. Umí pracovat s mnoha typy souborů (vtk, stl, trk, nrrd, fsm). Podpora ve všech hlavních prohlížečích. Jedná se o knihovnu, nikoli o komplexní software. Jednoduchý návod jak knihovnu použít je dostupný na stránkách projektu [1].

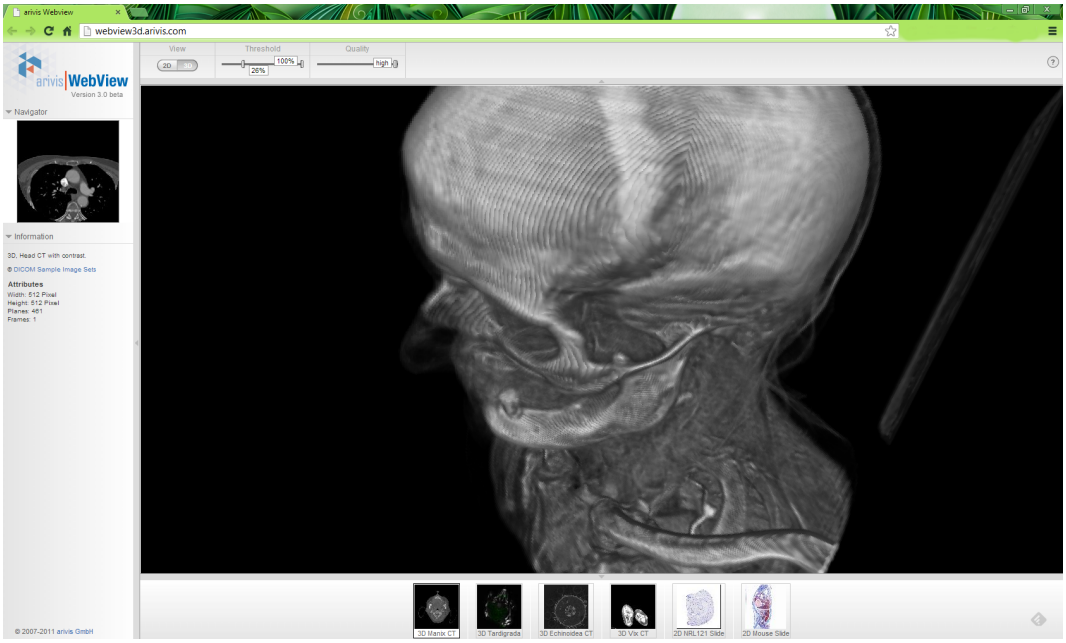
- VolumeRC [26] je experimentální open-sourcový projekt zaměřující se na vizualizaci dat ve webovém prohlížeči bez jakýchkoli pluginů. Volume rendering pomocí algoritmu ray casting je vidět zde[2]. Kvalita zobrazení však není příliš velká. Otázkou je, jak velké kvality lze dosáhnout s omezenými prostředky.

- Arivis WebView 3D[27] považuji za nejkomplexnější program z výše uvedených (pozn. jedná se pouze o můj subjektivní názor). Výrobce deklaruje práci s opravdu velkým množstvím dat. Graficky je na vysoké úrovni. Volume rendering také nemá nejhorší kvalitu. Aby byl ale program opravdu komplexní, tak zde chybí alespoň plně funkční multiplanární zobrazení (je zde pouze možnost zobrazovat původní řezy). Naopak velkou výhodou jsou nástroje jež program nabízí. Například měření vzdálenosti v datech. Ukázka programu je na obr. 5.1.

5.2 Známé problémy

Jedním z nejznámějších problémů, který podstatně snižuje využitelnost WebGL pro vizualizaci objemových dat ve webovém prohlížeči, je nedostupnost 3D textur. Jak bylo psáno v úvodních kapitolách, tato podpora není v OpenGL ES 2.0, ze kterého WebGL vychází zahrnuta. Neznamená to ale, že by byl volume rendering ve WebGL zapovězen. Jen není možné udělat v současné době tak výkonný vizualizační nástroj. Poměrně dobře jdou implementovat Texture-based metody, popřípadě (a tím se budeme zabývat v této práci) lze

[1] https://github.com/xtk/X
[2] http://www.volumerc.org/demos/volren/index.html



Obrázek 5.1: Program Arivis.

renderovat objemová data algoritmem ray casting, avšak i zde je třeba řešit mnoho problémů.

Dalším, avšak stále neřešitelným problémem, který vychází z podstaty běhu aplikací přímo ve webovém prohlížeči, je pomalá rychlost JavaScriptu ve srovnání s nativně kompilovanými jazyky. V dnešní době je značně zrychlený, ale pro takto náročné výpočty (například generování tří sad textur pro texture-based metody) jeho rychlost nestačí. Na běžné řízení GUI a hlavního vlákna programu (bez náročných výpočtů) je již rychlost zcela dostačující. Příprava dat se dá samozřejmě řešit předem, například před generováním textur ve všech třech osách, popřípadě využitím Native Client (experimentální projekt umožňující běh programů v nativním jazyce v prohlížeči[3]). Použití technologie Native Client je v současné době k dispozici pouze aplikacím distribuovaným přes Chrome Store.

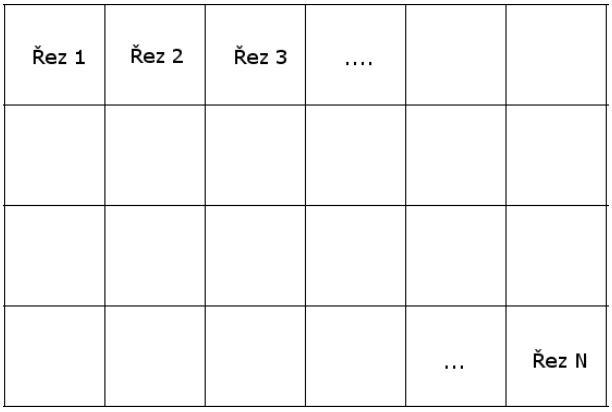
V neposlední řadě lze volumetrická data (často medicínsky zaměřená) považovat za velmi citlivá. Proto je třeba data dostatečně chránit. V cílové aplikaci je tento problém řešen tak, že data na server, na kterém je uložena aplikace, vůbec nejsou přenášena a nikdy s nimi server nepřijde do přímého styku. Data jsou vždy pouze u uživatele.

5.3 Řešení pomocí algoritmu Volume ray casting

Jak Ray casting funguje bylo popsáno v kapitole 4.5. Hlavním rozdílem je, že ve WebGL není podpora 3D textur. Musíme tedy do shaderu dostat data jinou formou. Jako ideální se nabízí využít fragment shaderu a data do něj předat jako velkou texturu (vize textury na obr. 5.2. Jak je vidět z obrázku, jednotlivé řezy dat jsou naskládány vedle sebe. Nevýhodou je značné snížení kvality výsledného obrazu, jelikož velikost textury, jenž je možné do fragment shaderu předat, je v dnešní době nejčastěji omezena na 4096x4096px. To není mnoho. V ostatních ohledech by mělo být řešení stejné (nebo podobné jako v OpenGL). Rozměry

[3] https://developers.google.com/native-client/quick-start?hl=cs

textury a počet řezů je třeba také předat do shaderu.



Obrázek 5.2: Reprezentace objemových dat velkou texturou.

5.4 Možné optimalizace

- Zpracovávat pouze data (voxely), které obsahují data s vysokou informační hodnotou - tím je myšleno přeskočení zpracovávání voxelů, které obsahují pouze informace o volném prostoru, nikoli o hmotě
- Zvýšení kvality obrazu kombinací více metod (teoreticky možná kombinace ray castingu a texture mappingu)

Kapitola 6

Návrh programu pro vizualizaci medicínských dat

Tato kapitola pojednává o volbě technologií a návrhu základních komponent aplikace, mezi které patří správa uživatelských účtů, zabezpečení dat, ukládání dat a návrh uživatelského rozhraní. Popisuje také základní koncept, jak bude aplikace fungovat.

6.1 Základní specifikace aplikace

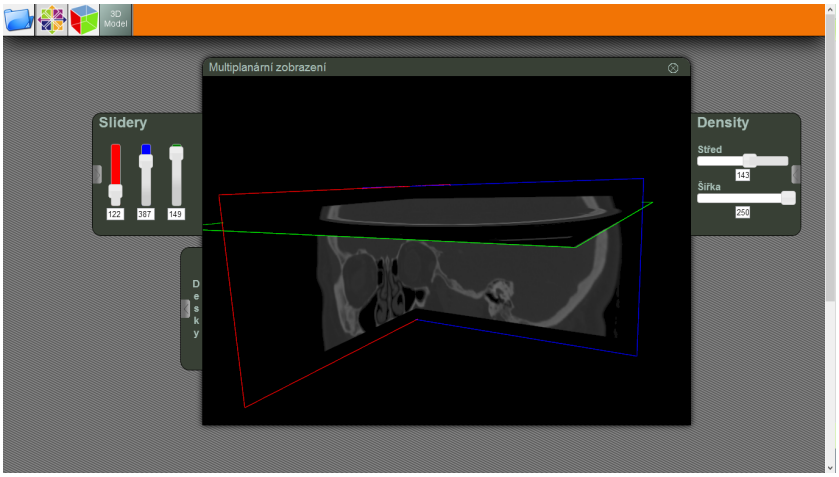
Výčet vlastností, které by měla výsledná aplikace mít, aby byla použitelná v praxi je uvedená v seznamu níže. Jedná se samozřejmě o opravdu minimální funkce, mělo by to však pro jednoduchou vizualizaci dat stačit. Jedna z klíčových vlastností, které ale ve výčtu chybí, je měření vzdálenosti v datech a načítání komprimovaných DICOM dat. Dále také generování 3D modelů některou z nepřímých renderovacích metod. To už ale v rámci diplomové práci zakomponováno nebude.

- Uživatelské účty (přihlašování přes Google účet)
- Propojení s externími úložišti (Google Drive, Dropbox, ..)
- Nahrávání souborů z lokálního disku
- Multiplanární zobrazení
- Volume Rendering (Ray casting)
- 2D zobrazení jednotlivých řezů
- Načítání nekomprimovaných DICOM dat
- Ukládání poznámek k datasetům
- Multijazyčnost (Čeština, Angličtina, Němčina)

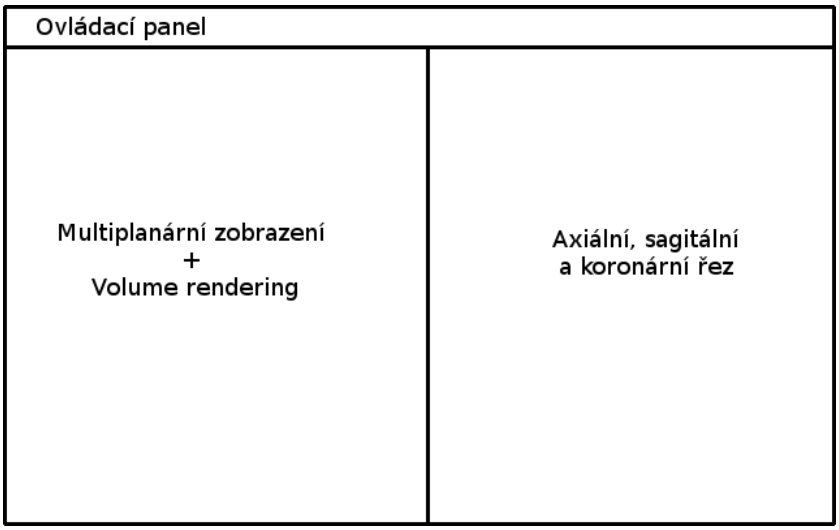
6.2 Návrh rozhraní

Je velmi složité navrhnout ideální a intuitivní rozhraní. Cílem bylo navrhnout rozhraní s maximální efektivitou ovládání a současně s co největší zobrazovací plochou. Implementovaný prototyp rozhraní je vidět na obr. 6.1. Po prvotních testech se ukázalo rozhraní

neefektivní. Uživatele mátlo zobrazení okna v okně. Po testech a konzultacích se rozhraní značně zjednodušilo. Nová vize je znázorněna na obr. 6.2. Finální verze bude jistě vypadat trochu jinak, ale použití principu rozdělení obrazovky je konečné.



Obrázek 6.1: Prototyp rozhraní.



Obrázek 6.2: Nový koncept rozhraní.

Rozložení oken je navrženo s pevným rozmístěním. Uživatel nebude mít možnost ho jakkoli nastavovat. Jediná nastavitelná položka je posuvník mezi pravou a levou zobrazovací částí. Uživatel tak může definovat poměr zobrazení jednotlivých oken. V levé části lze přepínat mezi multiplanárním a volumetrickým zobrazením. V pravé části je pak 2D zobrazení jednotlivých řezů (koronální, sagitální a axiální řez).

Jednotlivé ovládací prvky jsou řešeny pomocí vyjíždějících nabídek. Tento přístup šetří cenný pracovní prostor, jelikož si je uživatel vysune pouze v případě potřeby (to je u některých funkcí velmi zřídka). Jak výsuvná nabídka vypadá, je na obr. 6.1. Tyto nabídky zůstaly po rozsáhlých změnách rozhraní zachovány.

6.3 Uživatelské účty

Přihlašování přes Google účet
Jelikož aplikace pracuje s citlivými daty, bylo třeba dostatečně zabezpečit správu uživatelů. Z toho důvodu je použito již existující a prověřené řešení. Přihlašování pomocí účtu od společnosti Google. Velkou výhodou této volby je současné napojení ostatních služeb, které Google poskytuje, přímo na vyvíjenou aplikaci. Jedním z těch, jenž budou implementovány, je připojení Google disku. Ten je každému uživateli s Google účtem automaticky vytvořen. Nemluvě o zabezpečení přenosu mezi Google diskem a aplikací, které je zabezpečeno na vysoké úrovni.

Google API používá k autentizaci a autorizaci protokol OAuth 2.0, který umožňuje ověření uživatele u aplikací třetích stran, aniž by uživatel musel své přihlašovací údaje aplikacím sdělovat. Vice naleznete na stránkách Googlu[1] a webu OAuth[2].

Nutnost použití účtu od společnosti Googlu nepovažuji za limitující. Uživatel, který tento účet vlastní, se nemusí zdržovat registrací. Uživatel, který účet nemá a musí si účet vytvořit, by ztratil stejné množství času při registraci na serveru aplikace. Klady tudíž jednoznačně převažují.

6.4 Datová úložiště

Nedílnou součástí aplikace je prostor pro uložení dat. Standardním řešením je vytvoření místa pro uživatelova data na straně serveru, který danou službu poskytuje. Toto řešení bylo zhodnoceno jako příliš nákladné. Aplikace se může stát populární a velmi používanou. Z tohoto důvodu je cíleno na minimální provozní náklady, což by při použití standardního scénáře nebylo reálné.

Rozhodl jsem se tedy aplikaci plně integrovat s externími úložišti. Možnost načítání dat z lokálního disku je samozřejmostí. Jelikož není cílem práce data upravovat, ale pouze vizualizovat, umí aplikace data pouze číst nikoli zapisovat. V současné době bude aplikace schopna načítat data reprezentovaná sadou jednotlivých obrázků (PNG, JPEG). I tak jsem se ale rozhodl zmínit odstavec o oficiálně uznávaném standardu DICOM.

DICOM
Formát DICOM (Digital Imaging and Communications in Medicine)[28] je standard pro ukládání a distribuci medicínských dat. O zdrojových datech ukládá velké množství informací. Je velmi rozsáhlý. V rámci diplomové práce nebude standard podporován. Do budoucna se s podporou počítá. Jedná se o významnou část celého programu při praktickém využití aplikace. V datech jsou totiž kromě jména a příjmení uloženy i informace o počtu řezů, orientaci snímaného objektu, ale především rozměry voxelu snímaných dat. Znát rozměry voxelu je důležité, protože bez jejich znalosti není výsledné zobrazení ve správném poměru. Data budou deformovaná. Dočasným řešením vzniklé situace je dialog, který se zobrazí těsně po načtení dat ze serveru. Do tohoto dialogu uživatel zadá potřebné údaje. V praxi by tento přístup použitelný nebyl, protože uživatel by nemusel rozměry voxelu znát. Na funkci zbytku programu to ale nemá vliv.

[1] https://developers.google.com/accounts/docs/OAuth2Login
[2] http://oauth.net/

Úložiště

- **Google Drive**
  Primárním externím úložištěm je v aplikaci úložiště od Googlu. K tomuto kroku bylo přistoupeno z několika důvodů. Jedním z nich je automatické vytvoření datového úložiště pro každého uživatele při vytvoření účtu. Lze tedy s přítomností tohoto úložiště počítat. Dalším velkým plusem je zabezpečená komunikace mezi datovým úložištěm a mou aplikací. Ta je na velmi vysoké úrovni. Samozřejmostí je přenos pomocí SSL a omezená doba platnosti odkazu na soubor.

  Nevýhodou Google Drive API je omezený počet požadavků aplikace na Google disk za den. Počet požadavků se počítá za všechny uživatele dohromady. V současné době je omezen na 500 000 požadavků v jednom dni. K eliminaci tohoto omezení není použit přímý přístup k Google disku. Existuje řešení, které velmi redukuje počet reálných požadavků na službu. Jedná se o Google Picker[3]. Díky němu je možné stahovat velké množství souborů a vyhnout se tak přesažení limitů.

  Nevýhodou Google Pickeru je nemožnost ukládat na server uživatelská nastavení. V budoucnosti by stálo za zvážení vytvořit hybridní komunikaci, při které by se pro stahování dat používal Google Picker a pro ukládání drobných dat (nepříliš častý upload) využít plný přístup k dané službě.

- **Dropbox**
  V současné době (Duben 2013) existuje na Dropbox poměrně pěkné API. Je zde podobný nástroj jako Google Picker. Bohužel v základní formě lze označit vždy pouze jeden soubor. To je nevyhovující. Pak existuje možnost přístupu přes klasické Core API. To vyžaduje naprogramování vlastního modulu. V rámci diplomové práce tato vlastnost implementována nebude. V dalším vývoji aplikace se s implementací modulu počítá.

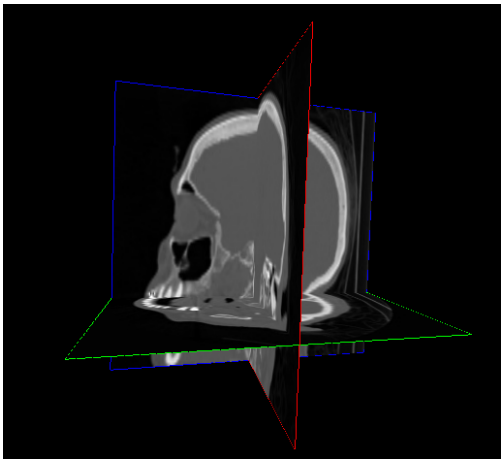
- **Lokální úložiště**
  Možnost načítání dat z lokálního disku je klíčové. Načtení dat je velmi rychlé a bezpečné. Nedochází k žádnému přenosu dat. K načítání dat bude použito File API[4]. Jedná se poměrně o novou funkci standardu HTML5. Načtení dat z lokálního úložiště umožňuje práci offline. Rozhraní podporuje nahrávání mnoha souborů. Pomocí tohoto API však nelze jakékoli soubory ukládat na disk. Je to pochopitelné, jelikož bezpečnostní rizika manipulace se soubory jsou příliš velká. Existuje způsob, jak této funkce dosáhnout. V případě vydání oficiální aplikace přes Chrome obchod je možné vyžádat si v manifestu, který popisuje požadavky aplikace, povolení manipulovat se soubory na disku. V tomto případě je to bezpečné, jelikož aplikace projde testováním na straně Googlu.

6.5 Multiplanární zobrazení

Multiplanární zobrazení je naprosto nezbytná součást jednoduché aplikace pro vizualizaci medicínských dat. Podrobněji se o něm rozepisovat nebudu. Základní principy jsou vysvětleny v mé bakalářské práci[5]. V rámci diplomové práce je tento modul třeba přepsat, jelikož

[3] https://developers.google.com/picker/
[4] http://www.w3.org/TR/FileAPI/
[5] https://wis.fit.vutbr.cz/FIT/st/rp.php/rp/2010/BP/11810.pdf

framework, ve kterém je modul zapsán, zanikl. Zároveň je nutné implementovat optimalizace, aby bylo vykreslování použitelné v praxi (dostatečně plynulé). Příklad multiplanárního zobrazení je vyobrazen na obr. 6.3.



Obrázek 6.3: Multiplanární zobrazení dat.

6.6 Zobrazení jednotlivých řezů

Ač se tato funkce může zdát jako duplicitní k multiplanárnímu zobrazení, není tomu tak. Jedná se o velmi podstatnou věc. Lékaři jsou zvyklí používat klasický 2D pohled na data. Z implementačního hlediska se jedná o pouhé mapování vygenerovaných textur (stejným způsobem jako generujeme textury pro multiplanární zobrazení). Tato funkce bude ve výsledném programu zcela jistě zahrnuta.

6.7 Volume rendering

Volumetrické zobrazení je stěžejní vlastností celé aplikace. Vzhledem k vysoké výpočetní náročnosti a nedostatečné rychlosti interpretace JavaScriptu bude nutné dělat jisté kompromisy. Zmíněny budou i optimalizace, které by velmi zrychlily renderování jednoho snímku, ale v rámci DP již všechny implementovány nebudou.

Použití metody Volume ray casting
Volumetrické zobrazení bude implementováno pomocí algoritmu volume ray casting. Jedná se o jedno z nejpřesnějších a nejdůvěryhodnějších zobrazení. Nepoužívají se žádná 3D primitiva. Výstup je čistě 2D obraz. Základní princip algoritmu je popsán v sekci přímých renderovacích technik 4.5. Algoritmus umožňuje maximální využití informační hodnoty dat. Nespornou výhodou je zobrazení průhledných materiálů. Nepřijdeme tak například o znázornění tekutin ve výsledném obraze. Na základě odstínu jednotlivých voxelů jsme schopni odlišit jednotlivé materiály a tím vizualizaci umocnit. Mapování barev na jednotlivé voxely je popsáno níže.

Typy zobrazení
Aby byla aplikace reálně použitelná, je třeba podporovat více typů zobrazení. Mezi nejzá-

kladnější patří rentgenové zobrazení (X-ray), projekce maximální intenzity (MIP) a obarvování pomocí mapovací funkce (Shading). Aby bylo ovládání aplikace intuitivnější, budou přednastaveny mapovací funkce pro zvýraznění tvrdých a měkkých tkání. Uživatel si bude moci libovolně navrhnout vlastní mapovací funkci v generátoru.

- **X-Ray** neboli rentgenové zobrazení patří mezi naprostou nutnost. Z volumetrických dat vytváří klasické rentgenové snímky, se kterými jsou lékaři zvyklí pracovat. Výhodou takto interaktivního zobrazení je, že si lékař může vygenerovat snímek z jakéhokoli úhlu.

- **MIP** (Maximum intensity projection) neboli zobrazení maximální intenzity spočívá v zobrazení voxelu, jehož intenzita je při průchodu paprsku tělesem maximální. Toto zobrazení vyniká rychlostí svého výpočtu. Jeho nevýhodou je ztráta informace o hloubce předmětu. Mohlo by tedy dojít k chybné interpretaci dat. Tomu je ale zabráněno možností s objektem otáčet. Při otáčení objektu pozorovatel získá přesnější představu, kde se nález v datech nachází.

- **Shading** neboli obarvované zobrazení má velkou vypovídací hodnotu. Umožňuje zvýrazňovat jednotlivé typy tkání. Je tedy možné zobrazit si pouze kosti nebo zrekonstruovat měkké tkáně subjektu. Jednotlivým tkáním lze nastavovat i odlišnou průhlednost. Není tedy problém zobrazit tvrdé tkáně neprůhledně bílou barvou a měkkou tkáň červeně s částečnou průhledností. Z hlediska interpretace hloubky v datech je toto zobrazení nejintuitivnější.

  Barva výsledného pixelu se získává akumulací barev voxelů, kterými paprsek při průchodu objemovým tělesem prochází. Zvlášť se akumuluje výsledná barva a zvlášť průhlednost dle vzorce 6.1.

$$\begin{aligned} \hat{C}_i &= (1 - \hat{A}_{i-1})C_i + \hat{C}_{i-1} \\ \hat{A}_i &= (1 - \hat{A}_{i-1})A_i + \hat{A}_{i-1} \end{aligned} \tag{6.1}$$

  kde naakumulovaná barva/alpha je značena $\hat{C}/\hat{A}$. Aktuální barva/alpha voxelu je označena znaky C / A. Základní zobrazení bez stínů, ale obarvené, vznikne implementací vzorců 6.1.

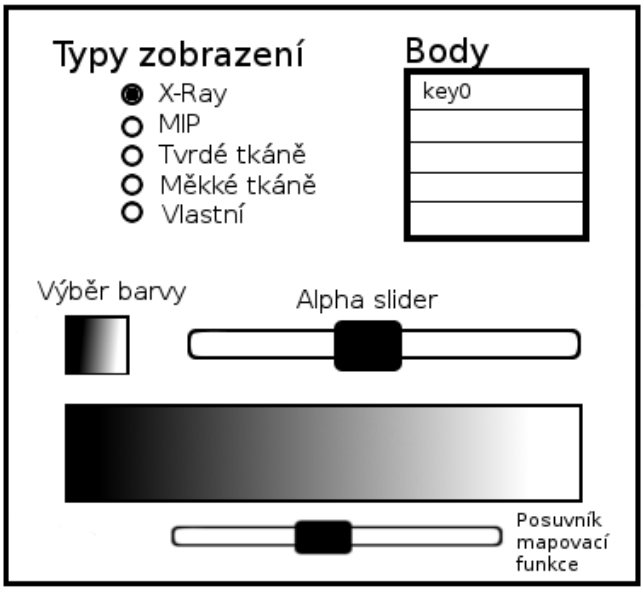
- **Vlastní nastavení** umožňuje manipulaci s nastavením mapovací funkce. Uživatel může přiřazovat barvu a průhlednost jednotlivým úrovním intenzit.

Generátor mapovací funkce

Jak bylo psáno v odstavci výše, lze jednotlivé voxely obarvovat dle jejich intensity rozmanitými barvami s různou průhledností. Aby aplikace věděla jak jednotlivé voxely obarvovat, je nutné jí dodat mapovací funkci. Vytvořit optimální mapovací funkci je velmi náročné. Navrhl jsem jednoduché rozhraní, které se maximálně snaží tuto tvorbu zjednodušit. Umožňuje ale vytvářet pouze jednoduché 1D funkce, které neumožňují brát v potaz rychlost změny intensity mezi jednotlivými voxely.

Na obrázku č. 6.4 je vidět základní vize, jak by mělo rozhraní generátoru vypadat. V horní části jsou přepínače, které umožňují vybrat mód zobrazení. V případě, že uživatel vybere X-ray nebo MIP, pak se zbytek nastavení generátoru nebere v potaz. V případě zvolení „Tvrdé tkáně“ nebo „Měkké tkáně“ se zbytek generátoru sám předvyplní. Uživatel ale bude

moci všechny parametry aktivně měnit. Při zvolení položky „Vlastní“ bude nastavení zcela na uživateli. Velmi důležitý je posuvník, který umožňuje posouvat celou mapovací funkci. Pomocí něj je možné dobře vytvořenou funkci namapovat i na data s jinak nasnímanými intensitami.



Obrázek 6.4: Návrh generátoru mapovací funkce.

Osvětlovací model

Osvětlení je podstatná část vizualizace, jelikož zvýrazňuje nerovnosti renderovaného povrchu.

Pro výpočet osvětlení bude použit Lambertův osvětlovací model, jenž byl podrobněji popsán v teoretické části. Pozice světla v prostoru bude konstantní.

Hlavním problémem, který je třeba vyřešit, je absence normál v datech. Ty určují, jak je povrch voxelu orientován vzhledem ke světlu a jak velké množství světla tudíž odráží. Normály lze nahradit gradienty. Ty se budou počítat v reálném čase na GPU, jelikož jejich předpočítání v JS je velmi pomalé (původní prototyp aplikace gradienty předpočítával).

Simulace 3D textury

Nahradit chybějící podporu 3D textur je výpočetně velmi náročné. Jak bylo řečeno v odstavci, který pojednává o přípravě dat (7.6), musí se z 3D textury vygenerovat odpovídající 2D textura, jenž bude obsahovat všechny řezy původní 3D textury rozprostřeny po celé své ploše.

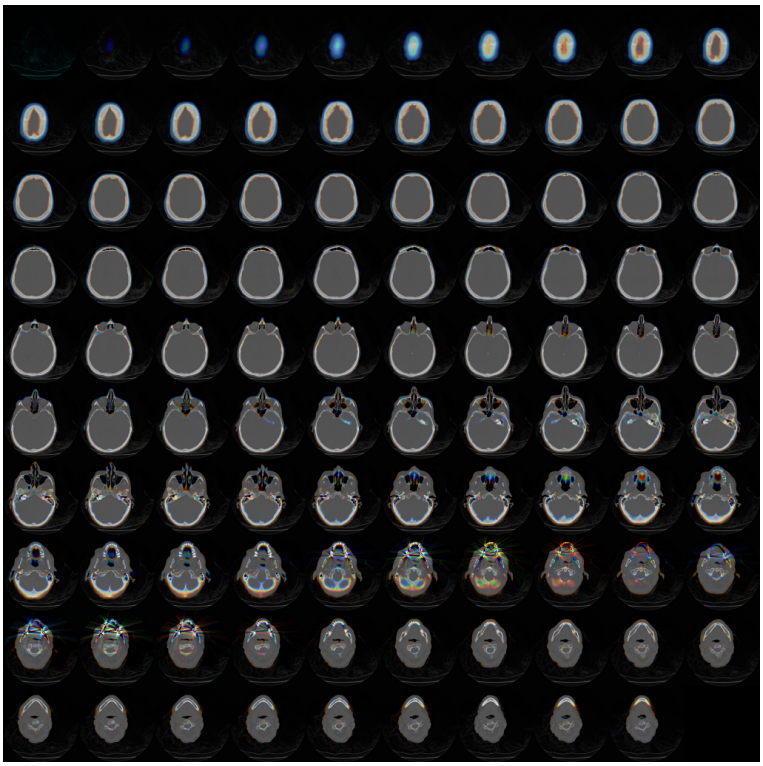
Nejpodstatnější na celém procesu je najít ideální poměr zmenšení řezu. Tato operace probíhá ve dvou fázích. Nejprve se spočítá přesný a zcela ideální poměr (připuštěny jsou pouze čtvercové řezy). Poměr je možno získat následujícím způsobem. Nejdříve se vypočte celková plocha, která je k dispozici (šířka $\times$ výška 2D textury). Tato plocha je pak vydělena celkovým počtem řezů. Odmocněný mezivýsledek reprezentuje ideální délku stěny.

Výsledný scale faktor lze získat jako poměr nové délky hrany ku staré. Celý výpočet je zapsán vzorcem.

$$scale = \frac{\sqrt{\frac{maxWidth \cdot maxHeight}{numOfCuts}}}{lengthOfEdge} \tag{6.2}$$

Bohužel situace není tak jednoduchá. Řezy se změněnou velikostí se zakreslují do *canvasu*. V canvasu je možné indexovat pouze pomocí celých čísel. Je tak nutné dosáhnout celočíselných rozměrů řezů a současně využít maximální plochu velké 2D textury. K tomuto je použit iterativní výpočet. Nejprve je třeba zaokrouhlit šířku a výšku jednoho řezu směrem dolů. Následně v cyklu rozšiřovat délku hrany o 10 pixelů a testovat, zda-li se řezy na texturu vejdou. Jelikož původní výpočet je poměrně přesný, nedochází k nijak dlouhému iterativnímu výpočtu. Průměrně je potřeba učinit 5 - 10 iterací. Algoritmus je vždy konečný. Při nalezení ideálních rozměrů řezu již další rozšíření není nutné. Může se ale nastat situace, kdy na okrajích velké textury zbývá pár pixelů. Textura se ořízne a přizpůsobí se opět na maximální rozlišení. Poměr stran jednotlivých řezů je sice narušen, to ale nehraje při dalších výpočtech roli, jelikož indexování na textuře je pouze relativní.

Těsně před zakreslením jednotlivých řezů do výsledné textury probíhá předzpracování dat. Cílem předzpracování je, aby byly do kanálů R a G každého pixelu v řezu uloženy intensity pixelů na totožných souřadnicích v okolních řezech (do R uložit předcházející a do G nadcházející intensity pixelu). Na obr. 6.5 je znázorněna optimalizovaná textura (lze si povšimnou využitých barevných složek v původně šedotónové textuře). V tuto chvíli jsou data připravena pro další zpracování v shaderu.



Obrázek 6.5: Optimalizovaná textura.

K indexování v rámci jednotlivých řezů slouží souřadnice x a y proměnné *posVolume*. Graficky je znázorněno na obr. 6.6. Není to ale triviální záležitost. Je třeba spočítat, kde

se nachází levý horní roh konkrétního řezu ve velké textuře (posun dx a dy). Tento bod lze spočíst na základě znalosti počtu řádků a sloupců v textuře. Podstatný je také index řezu. Informace o počtu řádků a sloupů, stejně tak informace o celkovém počtu řezů musí být do fragment shaderu předána pomocí uniformních proměnných. Souřadnice levého horního rohu je definována následujícím přepisem.

$$dx = \frac{mod(indexSlice, numColsInTexture)}{numColsInTexture} \tag{6.3}$$

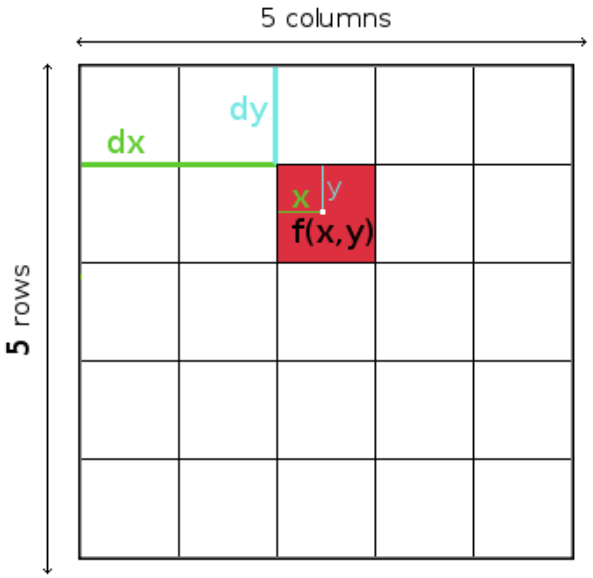
$$dy = \frac{floor(\frac{indexSlice}{numColsInTexture})}{numRowsInTexture} \tag{6.4}$$

Rovnice 6.3 a 6.4 určují posun, který je nutné následně připočítat k texturovacím souřad-

nicím udávajících polohu pixelu v 2D textuře. Finální texturovací souřadnice odpovídající přesné poloze pixelu, který se nachází na poloze hledaného voxelu, je počítána dle rovnic 6.5 a 6.6.

$$texCoordx = dx + \frac{x}{numColsInTexture} \tag{6.5}$$

$$texCoordy = dy + \frac{y}{numRowsInTexture} \tag{6.6}$$



Obrázek 6.6: Simulace 3D textury.

Výpočet gradientu

Gradient určuje změnu směru intenzity voxelu a jeho okolí. V aplikaci se gradient počítá pomocí středních diferencí (Central differences algorithm). Tento algoritmus je řazen mezi výpočetně nejrychlejší. Nepodává sice tak kvalitní výsledky jako výpočet gradientu pomocí Sobelova operátoru, ale pro nás je podstatná rychlost. Ačkoli je nutné počítat gradient ve

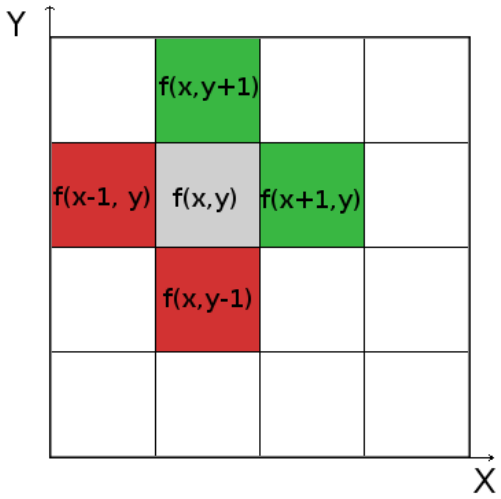
3D, pro jednoduchost je princip vysvětlen nad 2D daty. Výpočet nad 3D daty funguje na identickém principu, pouze se přidá jedna dimenze. Základem je spočítat rozdíly hodnot voxelů po obou stranách právě zkoumaného voxelu (ve směru dané osy). Vypočet je identický pro každou z os. Zapsáno pomocí vzorce 6.7 a 6.8.

$$g_x = \frac{f_{x+1,y} - f_{x-1,y}}{2} \tag{6.7}$$

$$g_y = \frac{f_{x,y+1} - f_{x,y-1}}{2} \tag{6.8}$$

$$G = g_x + g_y \tag{6.9}$$

Princip výpočtu je ilustrován na obr. 6.7. Šedou barvou je zvýrazněn aktuálně zpracovávaný pixel obrazu. Červenou / zelenou je pak označen předchozí / následující pixel v dané ose. Výsledný gradient je získán složením vektorů g_x a g_y. Příklad objemových dat se zvýrazněným gradientem je na obr. 6.8.



Obrázek 6.7: Výpočet gradientu - pixel.

Obrázek 6.8: Gradient objemových dat.

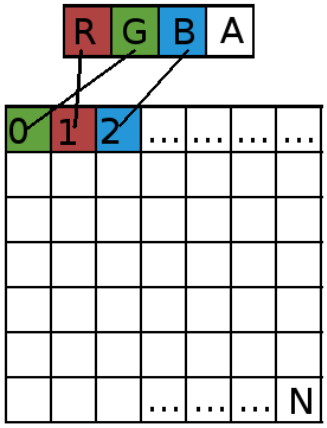
Optimalizace

Pro demonstrační účely v rámci DP nebudou velké optimalizace nutné. Následně ale bude zapotřebí celou aplikaci co nejvíce zefektivnit. Pro pohodlnou a efektivní práci je třeba dosáhnout relativně plynulé práce s daty (každý hodnotí plynulost jinak).

1. **Simulace 3D textury**

   Největší ztráta výkonu bude při simulování 3D textury pomocí 2D textury. K simulaci jednoho přístupu do 3D textury jsou třeba dva přístupy do 2D textury (proč tomu tak je, je vysvětleno v kapitole Implementace). To by ještě bylo přijatelné. Problém nastává při výpočtu osvětlovacího modelu. Při zjišťování gradientu voxelu musíme tradičně sáhnout do textury 6x (Central Differences). V případě simulace 3D textury je to 12x. Problém je také nutnost pohybu ve 2D textuře po velkých skocích (okolní řezy jsou daleko) a proto není dostatečně efektivní použití cache, kterou jinak grafická karta na přístup do textury používá. Existuje však možnost, jak minimalizovat přístupy do textury a práci tak velmi zefektivnit.

Každý voxel je reprezentován typem RGBA. Ve všech kanálech jsou však identické hodnoty. Bylo by tedy teoreticky možné, upravit výchozí texturu tak, aby ve zbývajících prázdných kanálech byly uloženy intensity předchozího a následujícího řezu. Princip je popsán na obrázku 6.9. Jednotlivé barevné kanály jsou barevně sladěny s řezy v 2D textuře. Aktuální řez, který je v oblasti zájmu, má hodnoty uložené v červené složce. Hodnota voxelu na stejných souřadnicích, ale o řez níže, má hodnotu uloženu v zelené složce. Ještě je třeba uchovat intensitu voxelu nadcházejícího řezu. Ta se uloží do modré složky. Alfa je nevyužita. Touto úpravou vstupní textury je následná simulaci 3D textury značně optimalizována. Již nebude nutné 2x přistupovat do textury na zcela odlišná místa. Postačí to pouze jednou.



Obrázek 6.9: Využití RGBA na reprezentaci okolních řezů.

2. **Délka kroku paprsku**

   Bez optimalizace prochází paprsek skrz objemové těleso s konstantní velikostí kroku, která je zadána uživatelem. To však není z hlediska výpočtu optimální. Velká část dat může být informačně nezajímavá, tudíž v této oblasti je možné mít krok velmi velký a výpočet tak značně urychlit. Celá situace bohužel není tak jednoduchá. Mohl by nastat případ, kdy zvolením příliš velkého kroku dojde k zanedbání významných dat. Jak tedy poznat optimální délku kroku?

   Objemová data jsou reprezentována 3D mřížkou voxelů. Jejím rozdělením na menší podbloky (v tomto případě na bloky 3x3), vznikne pomocná mřížka s třetinovými rozměry mřížky původní. Z každého podbloku objemového tělesa se vybere voxel s maximální intensitou a ta je následně zapsána do pomocné mřížky na odpovídající místo. Tuto pomocnou texturu si vygenerujeme pouze jednou. Při samotném renderování dat se primárně prochází menší texturou (krok je tedy značně velký). Při detekci důležité informace je velikost kroku okamžitě zmenšena a začne se klasicky akumulovat intensita z původní textury. Je tak bezpečně zaručeno, že omylem nedojde ke ztrátě významných dat a současně se výpočet značně urychlí.

Kapitola 7

Implementace aplikace

Kapitola detailně popisuje implementaci stěžejních modulů aplikace, dále rozebírá jak mezi sebou jednotlivé moduly komunikují a také se věnuje řešení volumetrického zobrazení. Celá aplikace je založena na knihovně Three.js, která velmi ulehčuje běžnou práci s 3D grafikou ve webovém prohlížeči.

7.1 Použité technologie

HTML 5 přináší klíčové technologie bez nichž by nebylo možné aplikaci implementovat. Nejdůležitějším prvkem, který je v aplikaci použit, je element CANVAS a jeho WebGL context. FileAPI, které přichází s HTML5, je použito k načítání souborů z lokálního disku.

WebGL je použito pro zobrazení 3D grafiky. Existují i jiné technologie, jak bylo popsáno v kapitole 2. Prototyp aplikace byl implementován v čistém WebGL, ale usoudil jsem, že použití knihovny je na místě, jelikož se aplikace příliš rozrůstala a mnoho věcí, jenž mi knihovna **Three.js** automaticky nabídne, bylo nutné ručně dopsat. Některé funkce (generování textur multiplanárního zobrazení) jsou stále napsány bez použití knihovny. Generování řezů je rychlejší, ačkoli s knihovnou Three.js už není rozdíl, v porovnání s knihovnou O3D, která byla pomalá, tak znatelný.

Javascript je použit pro řízení celého programu. Kromě jQuery nebylo užito žádné jiné knihovny pro práci s JavaScriptem. Do budoucna je v plánu aplikaci přepsat pomocí Google Closure, jelikož se jedná o velmi rozsáhlou aplikaci a implementace v čistém JavaScriptu nebyla nejlepší volbou.

GLSL je podporován pouze ve verzi 1.0. Tato verze bohužel obsahuje mnoho omezení. Jedním z nich jsou již mnohokrát zmíněné 3D textury. Jazyk GLSL jsem použil pro zapsání shaderu, jenž vykresluje volumetrické zobrazení.

CSS umožňuje vytvořit graficky poutavé uživatelské rozhraní. Zpřístupňuje velmi jemnou práci s prvky rozhraní. V nové verzi je dostupno i mnoho věcí, které bylo dříve nutné složitě implementovat.

jQuery je použito hlavně pro pokročilé prvku uživatelského rozhraní. Konkrétně se jedná o slidery, vysouvací nabídky a v neposlední řadě jQuery splitter.

Google Drive API obsluhuje práci s diskem Google Drive. Zpřístupňuje také Google picker. Ten je vhodné použít pro načítání dat, jelikož snižuje počet požadavků, které se započítávají do celkového limitu přístupů za den. Tyto limity jsou nastaveny na celou aplikaci.

7.2 Inicializace aplikace

Ihned po načtení aplikace, což zahrnuje načtení všech JavaScriptových modulů, souborů se styly a HTML šablonami, dochází k inicializace objektu *MainWatcher*. Jelikož grafické rozhraní není provázáno s jednotlivými moduly, musí objekt *MainWatcher* tato provázání vytvořit. Nastavuje také rozměry aplikace dle rozměrů stránky. JQuery Splitter, na kterém je celé rozhraní založeno, musí tyto parametry přesně znát. Dále je nutné nastavit všechny stavové proměnné, dle kterých se řídí ostatní moduly.

Jakmile je nastaveno uživatelské rozhraní, objekt inicializuje moduly pro načítání dat. V této fázi dochází ke kontrole, zda-li je aplikace online a je možné přistupovat k externím úložištím.

Jako poslední volá objekt *MainWatcher* metodu *appStart()* objektu *UiFunctions*, která inicializuje ostatní moduly a spustí v každém modulu vykreslovací smyčku.

7.3 Běh programu

Po úspěšné inicializaci aplikace a spuštění jednotlivých podmodulů (tj. multiplanární zobrazení, volume rendering a 2D řezy) přebírá řízení programu opět *MainWatcher*. Jeho instanci mají všechny okolní moduly a mohou nahlížet na jeho stavové proměnné. Jeho hlavním úkolem je zajistit, aby nedocházelo ke kolizím. Tento objekt se tedy chová jako „sběrnice“, které všichni sdělují informace o svém stavu a distribuuje je dál.

V prvním prototypu aplikace byly moduly zcela nezávislé. Z důvodů optimalizace a zjednodušení bylo přistoupeno k drobné integraci. Aplikace se nyní tváří jako celek. Ve skutečnosti jsou však moduly velmi nezávislé. Integrace spočívá v provázanosti pomocí několika málo proměnných uvnitř objektu *MainWatcher*.

Moduly stále běží ve svých smyčkách (k naplánování se používá *requestAnimationFrame(handle)*). Nabízela by se možnost využít více vláken. Ideálně pro každý modul, který vykresluje na obrazovku, vlastní vlákno. V JavaScriptu to bohužel není moc možné. To je poměrně problém, jelikož při zvýšené zátěži může mít uživatelské rozhraní zhoršenou odezvu. Nabízí se řešení použít takzvané „*WebWorkers*“. Tyto objekty nahrazují vlákna. Bohužel jejich možnosti jsou velmi omezené. V případě mé aplikace nepřipadalo jejich použití pro hlavní moduly v úvahu, jelikož všechny moduly přistupují do jedné sdílené paměti (obsahující velká volumetrická data). Sdílená paměť není u WebWorkerů podporována. To jsem vyřešil tak, že moduly reálně renderují pouze v případě, kdy jim k tomu uživatel dá podnět. Ve zbytku času běží renderovací smyčka naprázdno. Toto zatížení je natolik malé, že se na odezvě rozhraní nijak neprojevuje.

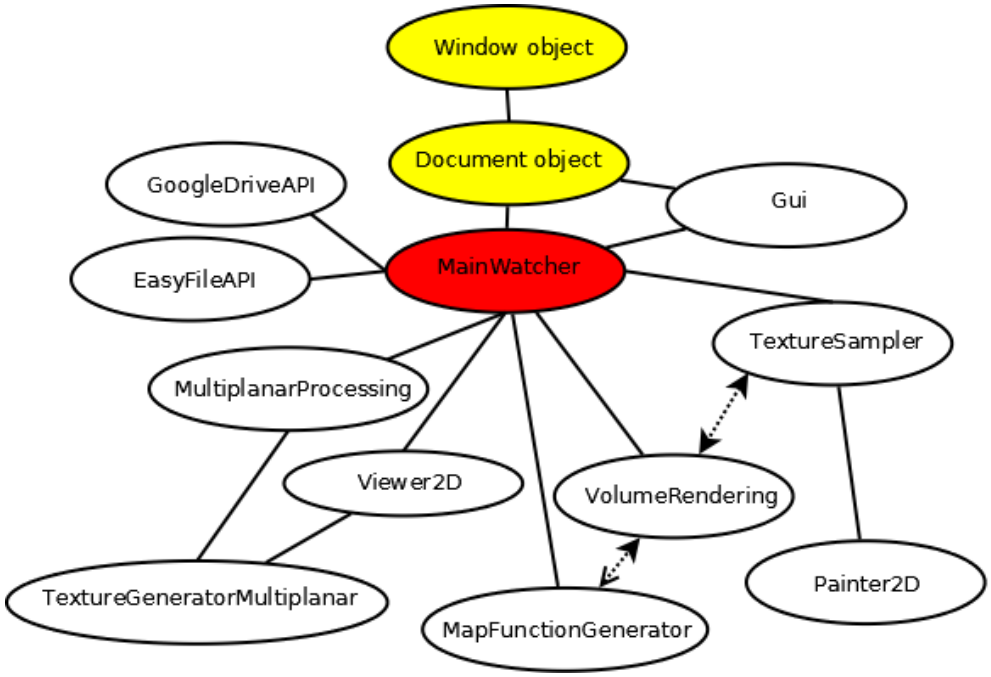
7.4 Architektura aplikace

Celá aplikace je implementována formou jednotlivých modulů. Řídícím a tedy i hlavním modulem je *MainWatcher*. Jak bylo psáno v přehledu jednotlivých komponent aplikace, slouží

k synchronizaci programu. Mezi další klíčové moduly patří: *GUI*, *GoogleDriveAPI*, *EasyFileAPI*, *MapFunctionGenerator*, *MultiplanarProcessing*, *TextureGeneratorMultiplanar*, *Viewer2D*, *VolumeRendering* a *TextureSampler*. Aplikace obsahuje více modulů. Ty jsou ale implementovány pouze jako pomocné a nebudu se o nich dále příliš zmiňovat. Jsou zahrnuty v grafu závislostí na obr. 7.1.

Níže je uveden zkrácený popis implementované funkčnosti jednotlivých modulů. U méně podstatných komponent je uveden krátký popis, k čemu slouží. Klíčové komponenty jsou následně rozepsány v samostatných blocích, kde je jim věnováno mnohem více prostoru.

- **MainWatcher** - Stará se inicializaci a řízení běhu programu.
- **GoogleDriveAPI** - Zajišťuje autentizaci na Google Drive. Inicializuje také rozhraní Google Pickeru (tj. nástroj na výběr souborů z úložiště). Jeho součástí je i dekomprese stažených dat a jejich nahrání do sdílené paměti (paměť využívána všemi moduly).
- **EasyFileAPI** - Umožňuje přístup na lokální úložiště. V současné době je možné pouze data načítat. Pomocí FileAPI, které modul využívá lze získat seznam odkazů na lokální soubory. Stejně jako *GoogleDriveAPI* obsahuje i tento objekt nástroje na dekompresi dat a jejich nahrání do sdílené paměti.
- **MultiplanarProcessing** - Stará se o projekci multiplanárního zobrazení. Návrh byl použit z bakalářské práce. V rámci diplomové práce jsem objekt kompletně reimplementoval a přizpůsobil do aktuálně vyvíjené aplikace. Více o návrhu a vlastnostech multiplanárního zobrazení lze nalézt v BP.
- **TextureGeneratorMultiplanar** - Slouží ke generování textur v jednotlivých osách. Tento objekt se používá při multiplanární projekci a zobrazené jednotlivých 2D řezů (sagitálního, axiálního a koronárního řezu).
- **Viewer2D** - Obstarává náhled nad daty z různých směrů (XY, YZ, XZ).
- **VolumeRendering** - Klíčový modul aplikace. Podrobně rozebrán v kapitole 7.7.
- **TextureSampler** - Tento objekt je schopen transformovat objemovou texturu do její 2D reprezentace. Při transformaci aplikuje na každý řez optimalizaci, kterou jsem popsal v odstavci 6.7. Výslednou texturu generuje v maximálním rozlišení, které daný grafický hardware podporuje (více v kapitole 7.6).
- **MapFunctionGenerator** - Objekt, která má na starosti generování mapovací funkce, dle parametrů od uživatele.



Obrázek 7.1: Propojení modulů aplikace.

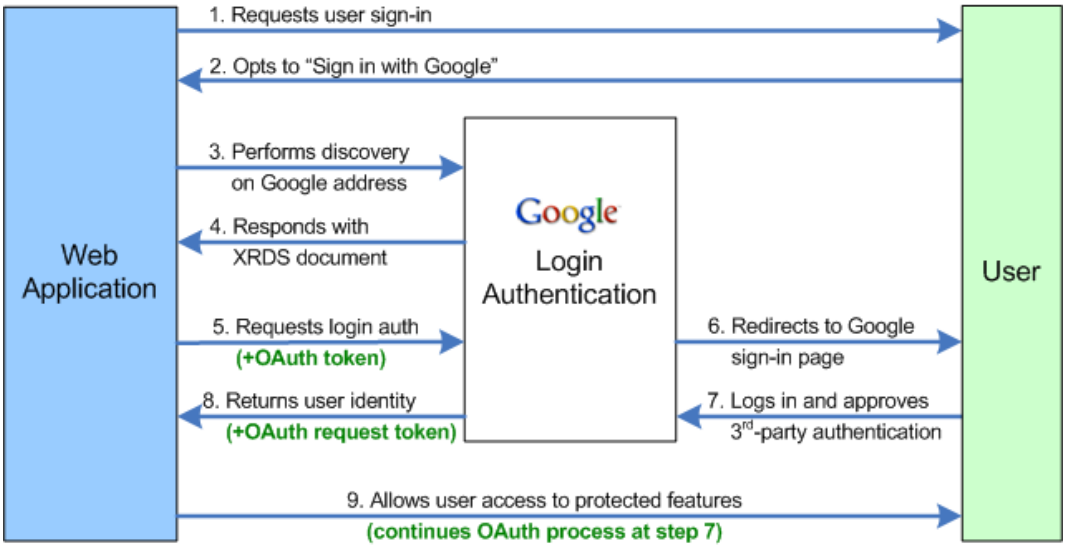
7.5 Uživatelské účty

Přihlašování do aplikace je řešeno přes účty Google. Průběh komunikace mezi aplikací, uživatelem a autentizační službou Googlu je znázorněn na obr. 7.2. Pro implementaci je použita knihovna Light OpenID[1]. Je ale dobré vědět, co se na pozadí děje.

Sekvence, která probíhá mezi aplikací, službou Googlu sloužící pro autentizaci uživatele a samotným uživatelem, je následující:

1. Aplikace požádá uživatele, aby se přihlásil
2. Ten si zvolí možnost přihlásit se pomocí účtu Googlu
3. Následně probíhá komunikace mezi aplikací a autentizační službou Googlu (dále jen služba)
4. Aplikace pošle službě požadavek na autentizaci uživatele.
5. Ta uživatele přesměruje na stránku „Google Federated Login“ (ta se otevře buď ve stejném okně nebo vyskočí pop-up okno). Na této stránce je uživatel požádán a zadání přihlašovacích údajů.
6. Uživatel musí schválit propojení jeho účtu s aplikací, do které se chce přihlásit (musí také odsouhlasit požadavky aplikace - čtení jména, přístup na disk a jiné - v závislosti na požadavcích aplikace).
7. Projde-li uživatel kontrolou služby, pak služba informuje aplikaci o úspěšné autentizaci.
8. Aplikace na základě úspěšné autentizace zpřístupňuje uživateli své funkce.

[1] https://code.google.com/p/lightopenid/



Obrázek 7.2: Průběh přihlašování uživatelů do webové aplikace [29].

7.6 Příprava dat

Vstupními daty jsou jednotlivé řezy, které jsou uloženy formou obrázků. Jakmile jsou jednotlivé řezy načteny, je třeba je seřadit dle názvu (načítání je asynchronní). Následně jsou zpracovány pomocí canvasu do 3D textury (canvas umožňuje velmi rychlou práci s obrazovými daty). Jeho nevýhodou je nutnost reprezentace barev typem RGBA (nadměrná spotřeba paměti). Jak je ale napsáno v kapitole 6.7, tyto duplicitní kanály barev lze využít k efektivní optimalizaci. V modulu *TextureSampler* jsou načtená data předzpracována, čímž lze účinně použít kanály R a G. Alfa kanál A je stále nevyužit. Předzpracování je sice výpočetně velmi náročné a trvá, provádí se ale pouze jednou při načtení dat. Ve výsledku je to velmi výhodné, jelikož dochází k velkému zvýšení rychlosti. Více informací lze nalézt v kapitole vyhodnocení.

Po načtení surových dat a zadání rozměru voxelu se aktivuje metoda *startRendering* objektu *MainWatcher*. První věc, kterou provede, je volání metody *start* objektu *TextureSampler*. Vstupními parametry jsou rozměry 3D textury a rozlišení textury, kterou má objekt produkovat. Velikost textury je zadána dle informací z HW grafické karty (objekt se snaží generovat texturu o maximálním možném rozlišení).

Generování 2D textury probíhá ve více krocích. Celé je řízeno metodou *transformCubeToTexture*, která postupně prochází objemové těleso řez po řezu a zakresluje je do velké 2D textury, která ve výsledku obsahuje všechny řezy naskládané v pravidelné mřížce. Ještě předtím než jsou všechny řezy do mřížky zakresleny, je třeba zjistit co nejoptimálnější rozměry řezů tak, aby se na výslednou texturu vešly všechny, ale byla zachována co nejvyšší kvalita. Podrobněji je vytváření textury rozepsáno v odstavci 7.7. Jakmile je textura vygenerována, je o výsledku informován *MainWatcher*. Ten distribuuje tuto informaci do modulu *VolumeRendering*.

7.7 Volume rendering

Základní princip volume renderingu byl vysvětlen v sekci 4.5. Princip je v základě poměrně jednoduchý. Je ale třeba řešit mnoho dílčích problémů. Nejpodstatnější je, kde a pod jakým úhlem vstupuje paprsek vyslaný z kamery skrze zobrazovací rovinu do objemového tělesa. Abychom měli přehled o poloze tělesa vzhledem ke kameře, je nutné ho ohraničit obalovým tělesem (tzv. bounding box). Základem je vytvořit těleso, které přesně kopíruje rozměry vstupních dat (započítán i relativní scale vzhledem k rozměrům jednoho voxelu). Stěny tělesa musí být obarveny tak, aby barvy stěn reprezentovali celý barevný prostor RGB.

Hodnoty mezi jednotlivými vrcholy se lineárně interpolují. Pomocí této techniky je možné přesně zjistit, kde do objemového tělesa vstupuje paprsek vyslaný z kamery a jakým směrem se paprsek v objemovém tělese pohybuje.

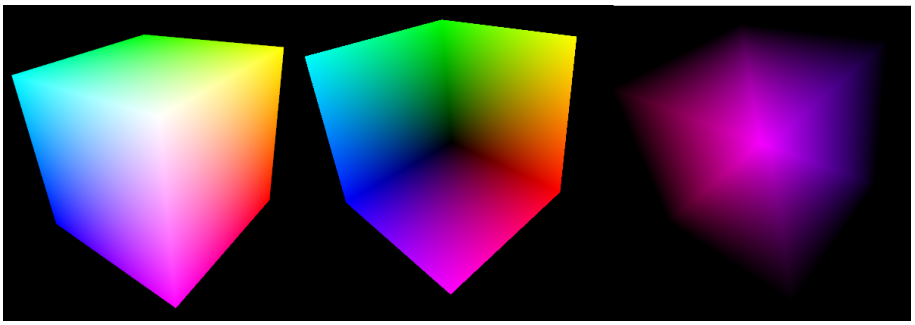
Při renderování scény jsou nejdříve vykresleny stěny, které nejsou vidět. V Three.js je třeba nastavit renderer příkazem

```
renderer.setFaceCulling(THREE.CullFaceFront)
```

Zadní stěny vykreslíme do framebufferu. Poté je nutno vykreslit celou scénu znova s ořezanými neviditelnými stranami.

```
renderer.setFaceCulling(THREE.CullFaceBack)
```

Neviditelné stěny z předchozího vykreslení jsou následně předány do fragment shaderu jako textura. Směr paprsků z kamery do scény lze zjistit odečtením hodnot překrývajících se barev předních a zadních stěn. Výsledné směry paprsků jsou znázorněny na obr. 7.3 vpravo.



Obrázek 7.3: Bounding box.

Jakmile je znám směr paprsku z kamery, stačí vzít aktuální barvu pixelu (souřadnice voxelu), skrze který byl do scény paprsek vyslán. Barva pixelu určuje přesnou pozici voxelu, skrze který paprsek vyslaný z kamery vstupuje do objemového tělesa. Délka kroku je závislá na požadované kvalitě (čím menší krok uživatel zvolí, tím přesnější je výsledek). Finální intensitu pixelu lze získat postupnou akumulací barev jednotlivých voxelů v daném směru s daným krokem (implementováné *for* cyklem).

Princip výpočtu délky kroku je popsán ve zdrojovém kódu 1. Zdrojový kód detailně popisuje výpočet pro získání směru kroku i jeho délky. Následně vysvětluje, jak dochází k pohybu paprsku skrze těleso. Kód zviditelňuje podstatné části a je neúplný. Základní principy potřebné pro implementaci volume renderingu jsou ale vyobrazeny.

X-Ray zobrazení
Rentgenové zobrazení spočívá ve vypočtení průměrné intensity všech voxelů, které se ve směru paprsku nachází. Procházení všech voxelů je výpočetně velmi náročné. Základní princip je znázorněn algoritmem zapsaným v příkladě 2.

MIP zobrazení
Projekce maximální intenzity je zobrazení, při kterém je výsledný voxel vždy ten, jehož intenzita je největší ze všech voxelů v dráze paprsku. Princip je zapsán v programu 3.

Shading
Barvené zobrazení bez stínů patří mezi nejméně náročné. Jakmile je výsledná barva zcela neprůhledná, dochází k zastavení kumulativního výpočtu. Při použití stínů je výpočet velmi náročný, jelikož na výpočet gradientu je třeba do textury přistupovat 6x (bez optimalizace 12x).

Pro získání opravdu kvalitních výsledků je třeba vytvořit osvětlení scény. Postačí nejzákladnější Lambertův osvětlovací model. Pozici světla ve scéně je určena relativně ke kameře (to z toho důvodu, že v Three.js se při ovládání ve skutečnosti neotáčí těleso, ale kamera okolo něj). Je-li připojeno světlo ke kameře, chová se ve scéně konstantně. Výpočet osvětlení je implementován dle popisu v odstavci 6.7.

Generátor mapovací funkce
Modul pro generování mapovací funkce se skládá z více objektů. Hlavní objekt je nazván *MapFunctionGenerator*. Pomocné podmoduly jsou pak nazvány *OptGroup*, *TriangleMark* a *KeyPoint*. Modul umožňuje uživateli navrhování vlastních mapovacích funkcí a je implementován následovně.

Nejdříve dojde k inicializaci uživatelského rozhraní a aktivací defaultně nastaveného zobrazení X-Ray. Výběr zobrazení je implementován pomocí „Select boxu“. Aktuální hodnota selectboxu je předávána do shaderu pomocí uniformní proměnné. Shader se dle toho následně rozhoduje, které zobrazení bude aktivní (MIP, X-Ray, Shading). Pro zobrazení klíčových bodů je použit rovněž „Select box“, ale v jiném zobrazení (rozbalené). Klíčová část rozhraní (zobrazení textury) je reprezentována canvasem.

Jednotlivé body uživatel přídává kliknutím myší do canvasu. V místě kliku se vytvoří bod a přídá se do seznamu klíčových bodů. Je mu nastavena barva dle nastavení color pickeru a alpha dle sliderů. Klikne-li uživatel na již existující bod a drží tlačítko myši stisknuté, může pak s bodem pohybovat. Změna bodu je detekována dvojitým poklepáním na bod (ten zmodrá). V této chvíli je jeho nastavení synchronizováno s aktuálním nastavením barvy a alphy. Všechny změny v klíčových bodech se okamžitě projevují na zobrazení objemového tělesa.

Objekt *MapFunctionGenerator* po jakékoli změně kontroluje, zda-li nedošlo ke změně seznamu klíčových bodů. Jakmile zjistí, že ke změně došlo, okamžitě dle bodů generuje novou mapovací funkci a vykreslí ji do canvasu jako texturu. Pro vygenerování nové textury používá metodu *generateMapTexture*. Ta funguje poměrně jednoduše. Pole bodů seřadí dle pozice v canvasu (body leží v intervalu $(0, 255)$). Pak postupně interpoluje vždy mezi dvěma sousedními body barvu a průhlednost.

O seznam klíčových bodů se stará objekt *OptGroup*. Vykreslování jednotlivých trojúhelníků reprezentující body v canvasu obstarává objekt *TriangleMark*. Aby byla práce co nejjasnější, je každý bod reprezentován objektem *KeyPoint*. Tento objekt uchovává všechny podstatné informace o bodu (pozice, barva, průhlednost a další).

Program 1 Výpočet barvy pixelu skrze který je vyslán paprsek

```
vec3 rayDirection = texture2D(backFaceCube, texCoo).xyz - varcolor.xyz;
float lenRay = length(rayDirection); //delka paprsku
vec3 normDir = normalize(rayDirection); // normalizovany smer paprsku
float d = qualitySteps; // quality steps určuje velikost kroku
vec3 step = normDir * d; // step určuje směr paprsku o konkrétní velikosti
float lenStep = length(step); // delka kroku
float accumulatedLength = 0.0; // urcuje dosavadni delku paprsku

for(int i = 1; i < maxStep; ++i) {
  vec4 intensityVoxel = getVolumeSample(pos);
  finalColor = accumColor(intensityVoxel, finalColor);
  pos += step; //posouvá na další vzorek
  accumulatedLength += qualitySteps; // akumuluje jiz projdutou delku paprsku

  /* pruchod paprsku telesem se zastavuje ve chvíli, kdy je paprsek
   * dostatečně dlouhý, nebo je barva jiz zcela neprůhledná
   */
  if(accumulatedLength >= lenRay || accumulatedColor.a > 1.0) {
    break;
  }
}
```

Program 2 Výpočet X-Ray zobrazení

```
vec4 accumColorXray(intensityVoxel, accumulatedColor) {
        vec4 voxelColored = vec4(intensityVoxel);
        vec4 sample = vec4(0.0, 0.0, 0.0, 0.0);
        sample.a = voxelColored.a * qualitySteps);
        sample.rgb = voxelColored.rgb * sample.a ;

        // akumulace barvy a alfy
        accumulatedColor.rgb += (1.0 - accumulatedColor.a) *
                                lightFactor * sample.rgb;
        accumulatedColor.a += sample.a;
        return accumulatedColor;
}
```

Program 3 Výpočet MIP zobrazení

```
vec4 accumColorMIP(intesintyVoxel, accumulatedColor) {
    vec4 voxelColored = vec4(intensityVoxel);
    if(accumulatedColor.a < voxelColored.a) {
        accumulatedColor.a =  voxelColored.a;
        accumulatedColor.rgb = voxelColored.rgb;
    }
}
```

Simulace 3D textury

Pro práci s velkou 2D texturou je v shaderu implementována funkce *getVolumeSample(vec3 posVolume)*, jejíž vstupní parametr je pozice v objemovém tělese. Její návratová hodnota je intensita voxelu, který se nacházel na daných souřadnicích. První věc, kterou je třeba zjistit je index řezu, jež je oblastí našeho zájmu. Index řezu lze zjistit vynásobením počtu řezů a z-ové souřadnice, která určuje pozici voxelu v tělese. Výsledkem většinou nebude celočíselné číslo. Řezy se však indexují celočíselně. Výpočet tedy musí být rozdělen na dva kroky. Nejdříve je potřeba zaokrouhlit výsledek vzniklý vynásobením počtu řezů a z-ové souřadnice směrem dolů na celé číslo. Intensitu, která je získána na souřadnicích (x,y) určených vstupním parametrem *posVolume*, je nutno si uložit. Následně se zvýší index řezu o 1 a přečte se intensita voxelu na zcela identickém místě jako v řezu s indexem o 1 méně. Jelikož jsou data předpřipravená a každý pixel velké textury obsahuje v G a B kanálech i intensity voxelů v okolních řezech, není třeba do textury znovu přistupovat. To značně zefektivňuje práci. K získání požadovaného výsledku, jenž bude co nejvíce odpovídat původním souřadnicím v tělese, je nutné tyto dvě intensity (R a G — B) z různých řezů interpolovat v poměru, jenž vyjadřuje poměr vzdálenosti pozice v objemovém tělese k oběma řezům. Interpolace je provedena funkci *mix*, která je dostupná v jazyce GLSL.

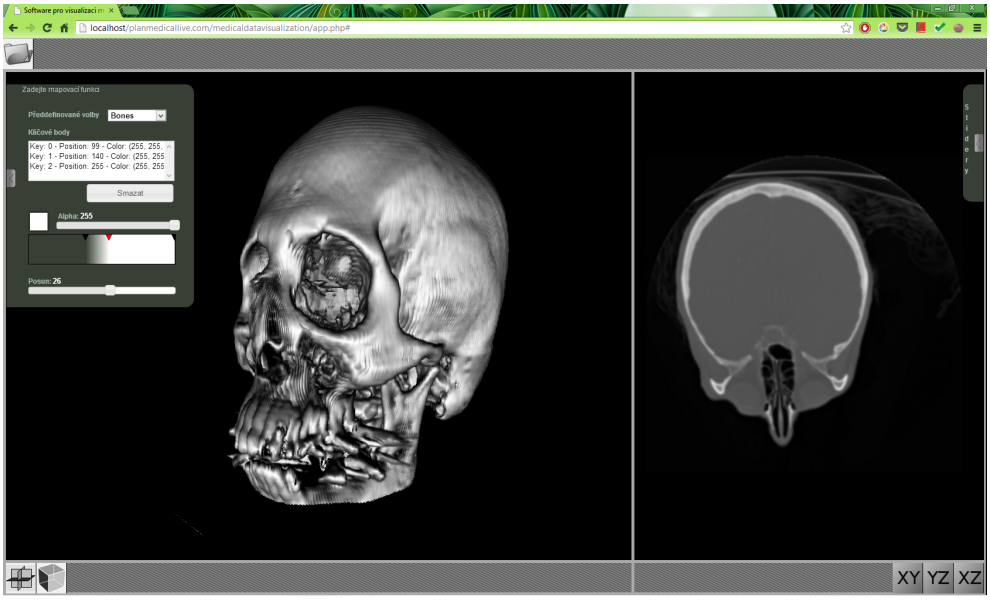
Kapitola 8

Vyhodnocení dosažených výsledků

Výsledkem diplomové práce je vizualizační program dostupný na adrese:

www.planmedicallive.com

Současný stav aplikace je na obr. 8.1. V levé části je zobrazeno volumetrické zobrazení. V pravé části pak pohled na data ve 2D.



Obrázek 8.1: Výsledná aplikace.

Testovací sestava
Všechny výsledky jsem naměřil na počítači s následující konfigurací (tabulka č. 8.1). Uvedená konfigurace umožňuje aplikaci dostatečně rychlou odezvu (hodnotit použitelnost je subjektivní, ale já ji považuji za dostatečnou). Pro běžná data postačuje konfigurace s 2GB RAM a slabší CPU. Po spuštění aplikace zabírá cca 50MB. Po provedení všech operací a poté co jsou v paměti ponechána pouze informačně významná data, zabírá u datasetu (542x542x178) cca 600MB. V průběhu zpracování však zabírá paměti více.

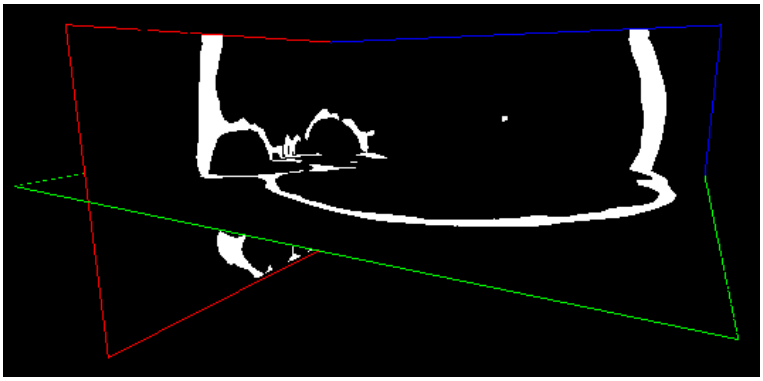
CPU	Intel Core i7 720QM (4 x 1.6GHz)
GPU	ATI Mobility Radeon HD5730 (1GB vlastní paměti)
RAM	8GB DDR3
OS	Windows 8
Prohlížeč	Google Chrome (Verze 26.0.1410.64 m)

Tabulka 8.1: Konfigurace testovacího PC.

Multiplanární zobrazení a 2D řezy
Grafický výstup multiplanárního zobrazení je identický jako v bakalářské práci, avšak po kompletní reimplementaci z frameworku O3D (který zanikl) do knihovny Three.js a implementaci nejkritičtějších částí do čistého WebGL se zvedla výkonnost a zvýšila plynulost vykreslování. Průměrná FPS pro různá rozlišení dat jsou uvedena v tabulce. Dalšího významného zrychlení bylo dosaženo implementací densitního okénka přímo v shaderu. Rychlost FPS je uvedena pro jeden řez. V současné době se současně generují řezy pro multiplanární zobrazení i náhled na konkrétní řezy odděleně. Moduly běží samostatně a vygenerované textury mezi sebou nesdílí. FPS tak může kolísat. Pro běžnou práci je to ale zcela dostačující. Lze si povšimnout, že FPS u multiplanárního zobrazení je nižší než u volumetrického. Ačkoli by se to mohlo zdát jako chyba, není tomu tak. U multiplanárního zobrazení se totiž textury na jednotlivých řezech generují na CPU pomocí JavaScriptu, aby nedošlo ke snížení kvality, jako k tomu dochází u volumetrického zobrazení. U volume renderingu po předzpracování dat už není CPU zatěžován a veškeré výpočty probíhají na GPU v shaderech.

Rozlišení dat (px)	**průměrné FPS / 1 řez**
256 × 256 × 176	24
542 × 524 × 176	18

Tabulka 8.2: Multiplanární zobrazení - statistiky



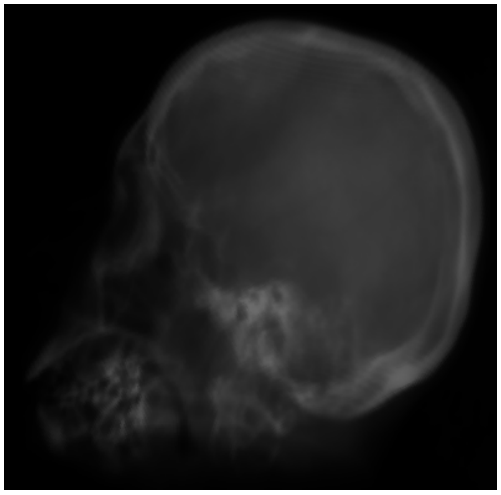
Obrázek 8.2: Multiplanární zobrazení.

Volumetrické zobrazení
Podařilo se implementovat plně funkční volumetrické zobrazení. Aplikace podporuje tři typy zobrazení (X-Ray, MIP a Shaded). Po řadě optimalizací jsem dosáhl poměrně vysokých FPS (více v tabulce). Kvalita zobrazení by mohla být vyšší (zmenšení velikosti kroku), pak ale FPS velmi klesá. Proto jsem kvalitu nastavil na přijatelnou tak, aby FPS bylo kolem dvaceti. Rychlost vykreslování také závisí na přiblížení objektu. Čím blíže, tím více rychlost klesá. Údaje v následující tabulce jsou zaznamenány při zoomu, jenž je nastaven ihned po spuštění aplikace.

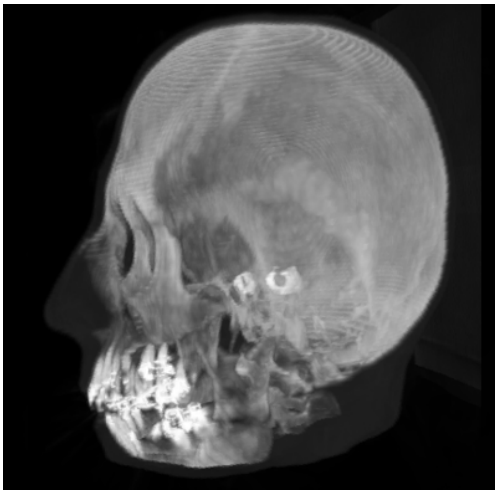
Typ zobrazení	**průměrné FPS**
X-Ray	30
MIP	32
Barevné (+ stíny)	20
Barevné (- stíny)	35

Tabulka 8.3: Volumetrické zobrazení (rozlišení obrazu: 400 × 400px).

Na obrázcích 8.3, 8.4, 8.5, 8.6 a 8.7 jsou znázorněny příklady jednotlivých zobrazení. Barvené zobrazení je zde ve třech variantách (tvrdé tkáně, měkké tkáně a kombinované tkáně). Fragmenty na objemovém tělese nejsou chybou, ale standardním projevem větší délky kroku paprsku.



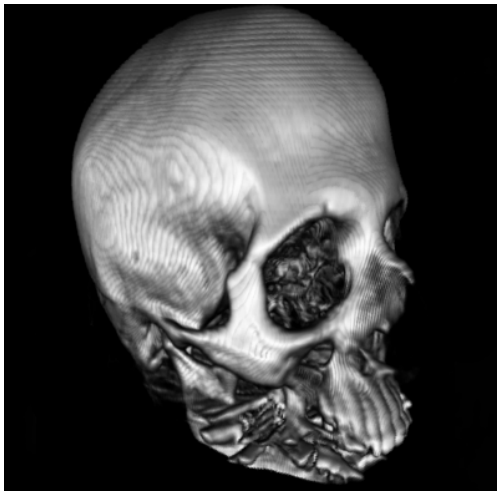
Obrázek 8.3: X-Ray.



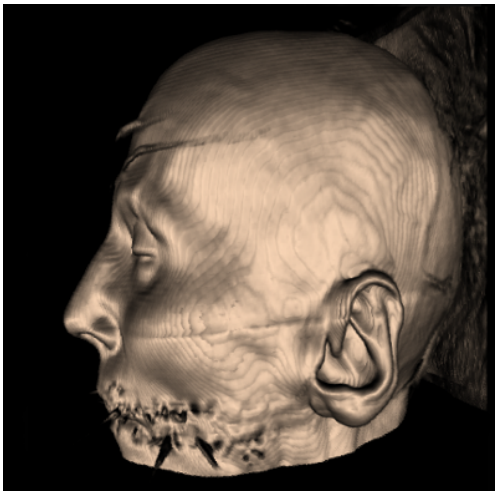
Obrázek 8.4: MIP.

V současné podobě je aplikace reálně použitelná pro jednoduchou vizualizaci. Jelikož ale dochází k neustálému vývoji knihovny Three.js a aplikace je velmi mladá, bylo by třeba k reálnému komerčnímu nasazení řádně otestovat všechny aspekty současné implementace. Ačkoli jsem mnoho variant testoval a mnohé opravil, občas dochází k náhodnému pádu aplikace. To ale nemusí být chyba v mé aplikaci, jelikož se v průběhu vývoje objevily některé chyby, které aktualizací na nejnovější verzi prohlížeče sami odezněly. Vyladění všech chyb by pravděpodobně zabralo podobný čas, jakou trvalo vytvoření současné aplikace. Z toho důvodu jsem se rozhodl jisté nedostatky akceptovat.

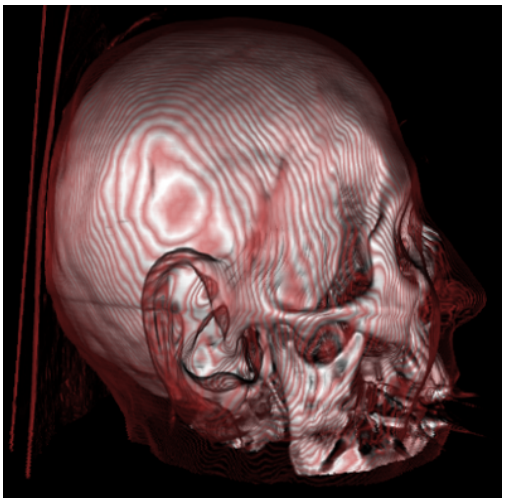
Do budoucna bych rád implementoval podporu DICOM souborů. Ta je velmi podstatná, protože bez ní nejsou k dispozici všechny potřebné informace pro zobrazení (rozměry vo-



Obrázek 8.5: Tvrdé tkáně.



Obrázek 8.6: Měkké tkáně.



Obrázek 8.7: Barvené zobrazení.

xelu, natočení dat při snímání). Dále je potřeba zvýšit rychlost volumetrického zobrazení. Jakmile bude dostatek výkonu, bude možné zmenšit délku kroku a tím zvýšit kvalitu. V další fázi vývoje je v plánu implementovat nástroje na měření vzdálenosti v datech a vkládání poznámek přímo do nich.

Kapitola 9

Závěr

Cílem práce bylo zjistit aktuální možnosti zobrazení pokročilé 3D scény v okně webového prohlížeče, prozkoumat aktuální knihovny a navrhnout aplikaci, která bude demonstrovat dostupné možnosti. Práce diskutuje o současných trendech a možnostech zobrazení 3D scény. Detailněji se zabývá popisem technologie WebGL a knihovnami, které se jí věnují. Navržená aplikace je implementována dle specifikace a prezentuje aktuální možnosti dostupných technologií.

Mezi klíčové vlastnosti softwaru patří multiplanární zobrazení, 2D pohledy a především volumetrické zobrazení. Velmi důležitým aspektem je napojení na Google Drive a to proto, že má-li být aplikace použitelná vždy a všude, je potřeba, aby byla vždy a všude dostupná také data. Primárně je vyvíjena pro prohlížeč Google Chrome.

Velká nepříjemnost byl zánik frameworku O3D, ve kterém byla implementována celá část zobrazující multiplanární zobrazení. Bylo nutné celý modul od základů reimplementovat s využitím knihovny Three.js, která se následně velmi osvědčila. Využil jsem toho pro provedení změn v původním návrhu aplikace na základě získaných zkušeností v bakalářské práci.

Při implementaci volumetrického zobrazení byl problém s chybějící reprezentací pro objemová data. V práci je podrobně rozepsáno, jak chybějící 3D texturu efektivně nahradit. Přestože není výsledná kvalita objemového zobrazení na tak vysoké úrovni jako v případě nativních aplikací, zdá se být dostatečná pro základní zobrazení, orientaci v datech a plánování. Při zvolení vyšší kvality rapidně klesala rychlost vykreslování.

Hlavním nedostatkem vývoje komplexních aplikací pro web je stále poměrně pomalý JavaScript, ačkoli v posledních letech dělá velké pokroky. Otázkou je, jak moc velké rezervy v optimalizaci ještě existují, protože se v poslední době začíná mluvit o jazyku Dart, který by měl JavaScript nahradit. Samotné renderování obsahu „canvasu“ JavaScript samozřejmě nijak neovlivňuje. Jak ale je zmíněno v práci, nejznatelnější zpomalení je při přípravě dat. Některé operace není ani možné v přiměřeném čase provádět.

V současné podobě je aplikace použitelná pro jednoduchou vizualizaci dat. Obsahuje množství nedostatků (např. chybějící podpora DICOM dat a dlouhodobé testování), ale to nevadí, protože ještě není plánováno aplikaci oficiálně vydávat. Jakmile budou tyto nedostatky odstraněny, věřím, že bude aplikace reálně užívána.

Aplikace byla prezentována na konferenci EEICT v roce 2013 pod názvem „3D medical data visualization in web browser“. Výtah z práce byl zařazen do sborníku a je k nalezení na webu.

Literatura

[1] Gothel, S. *JOGL* [online]. [cit. 2013-01-02]. Dostupné na: <https://jogamp.org/jogl/www/>.

[2] Adobe. *Stage3D* [online]. 2013 [cit. 2013-01-02]. Dostupné na: <http://www.adobe.com/devnet/flashplayer/stage3d.html>.

[3] Heuer, T. *A guide to Silverlight 3 new features* [online]. 2009-03-18 [cit. 2013-01-02]. Dostupné na: <http://timheuer.com/blog/archive/2009/03/18/silverlight-3-whats-new-a-guide.aspx>.

[4] Microsoft. *Silverligt: Out of Browser* [online]. [cit. 2013-01-02]. Dostupné na: <http://www.microsoft.com/silverlight/out-of-browser/>.

[5] Khronos Group. *WebGL Specification* [online]. 2011-01-12 [cit. 2013-01-04]. Dostupné na: <https://www.khronos.org/registry/webgl/specs/1.0/>.

[6] Forshaw, J., Stone, P. a Jordon, M. *WEBGL - MORE WEBGL SECURITY FLAWS* [online]. 2011-06-16 [cit. 2012-12-30]. Dostupné na: <http://www.contextis.com/research/blog/webgl-more-webgl-security-flaws/>.

[7] Khronos Group. *Getting a WebGL Implementation* [online]. 2012-12-07 [cit. 2012-12-26]. Dostupné na: <http://www.khronos.org/webgl/wiki/Getting_a_WebGL_Implementation>.

[8] IE WebGL Team. *IEWebGL* [online]. 2011 [cit. 2013-01-04]. Dostupné na: <http://www.iewebgl.com>.

[9] Khronos Group. *User Contributions - Frameworks* [online]. 2012-12-26 [cit. 2012-12-28]]. Dostupné na: <http://www.khronos.org/webgl/wiki/User_Contributions>.

[10] *Canvas 3d JS Library* [online]. 2010 [cit. 2013-01-03]. Dostupné na: <http://www.c3dl.org/>.

[11] Ambierra. *CopperLicht - fast WebGL JavaScript Engine* [online]. [cit. 2013-01-03]. Dostupné na: <http://www.ambiera.com/copperlicht/features.html>.

[12] Kay, L. *3D Scene Graph Engine for WebGL* [online]. 2009 [cit. 2013-01-03]. Dostupné na: <http://scenejs.org/>.

[13] *Three.js* [online]. [cit. 2013-01-03]. Dostupné na: <https://github.com/mrdoob/three.js>.

[14] Pinson, C. *OSG.JS* [online]. [cit. 2013-01-03]. Dostupné na: <http://osgjs.org/>.

[15] Kreylos, O. *Supersampling* [online]. 1999 - 2013 [cit. 2013-02-10]. Dostupné na: <http://idav.ucdavis.edu/ okreylos/ResDev/VolVis/LinkSupersampling.html>.

[16] MediTech Visual Aids. *MEDITECH VISUAL AIDS* [online]. 2013 [cit. 2013-05-19]. Dostupné na: <http://www.meditechvisualaids.com/3d-animations/single-gallery/2746502>.

[17] Brain Networks Laboratory. *Brain Networks Laboratory* [online]. 2004 [cit. 2013-05-18]. Dostupné na: <http://research.cs.tamu.edu/bnl/static/gallery3d.html>.

[18] Cengiz ÇELEBI Ömer. *Scientific Visualization and 3D Volume Rendering* [online]. [cit. 2012-12-30]. Dostupné na: <http://www.byclb.com/TR/Tutorials/volume_rendering/ch1_1.htm>.

[19] Pulo, K. *Isosurfaces* [online]. 2000-08-22 [cit. 2012-12-30]. Dostupné na: <http://www.kev.pulo.com.au/sv3/sv3_1999_assignment1/node4.html>.

[20] Anderson, B. *An Implementation of the Marching Cubes Algorithm* [online]. [cit. 2013-01-02]. Dostupné na: <http://www.cs.carleton.edu/cs_comps/0405/shape/marching_cubes.html>.

[21] Anderson, B. *An Implementation of the Marching Cubes* [online]. [cit. 2013-03-13]. Dostupné na: <http://www.cs.carleton.edu/cs_comps/0405/shape/>.

[22] University of Stuttgart. *Direct Volume Rendering* [online]. [cit. 2013-01-02]. Dostupné na: <http://cumbia.informatik.uni-stuttgart.de/ger/research/proj/ito/materials/VIS-Modules-06-Direct_Volume_Rendering.pdf>.

[23] Visualization Sciences Group. *Open Invertor developer zone: Volume Rendering Overview* [online]. 2010. Dostupné na: <http://oivdoc92.vsg3d.com/Users_Guide/VolumeViz/Volume_Rendering/Volume_Rendering_Overview.html>.

[24] Papademetris, X., Jackowski, M., Joshi, A. et al. *Bioimage Suite User's Manual*. 2008. Dostupné na: <http://bioimagesuite.yale.edu/manual/501_95522_bioimagesuite_manual.pdf>.

[25] The X Toolkit Developers. *The X Toolkit: WebGL for Scientific Visualization* [online]. 2012 [cit. 2013-01-03]. Dostupné na: <https://github.com/xtk/X>.

[26] Vicomtech IK4. *VolumeRC* [online]. 2011. Dostupné na: <http://www.volumerc.org/index.html>.

[27] Arivis. *Arivis WebView - Summary*. 2013. Dostupné na: <http://www.arivis.com/en/arivis-WebView/Summary>.

[28] NEMA. *Digital Imaging and Communications in Medicine* [online]. [cit. 2013-03-15]. Dostupné na: <http://dicom.nema.org/>.

[29] GOOGLE. *Federated Login for Google Account Users* [online]. 2013 [cit. 2013-05-18]. Dostupné na: <https://developers.google.com/accounts/docs/OpenID>.

Příloha A

Obsah CD

Optické médium, které je k práci přiloženo, obsahuje následující soubory a složky:

- /video - obsahuje video prezentující aplikaci a demonstruje její ovládání
- /plakat - obsahuje poster
- /screenshots - screenshoty prezentující vlastnosti aplikace
- /report - obsahuje technickou zprávu
- /src - zdrojové kódy webové aplikace
- /src/externalModules - externí knihovny
- /src/htmlModules - HTML šablony, ze kterých je složeno GUI aplikace
- /src/jquery - knihovna jQuery
- /src/models - 3D modely (definovány desky pro multiplanární zobrazení)
- /src/modules - JavaScriptové moduly (volume rendering + shadery, multiplanární zobrazení a další)
- /src/shaders - shadery pro multiplanární zobrazení
- /src/styles - CSS styly aplikace
- /src/textures - obsahuje všechny textury použité v aplikaci
- /app.php - hlavní GUI aplikace
- /index.php - přihlašovací obrazovka
- /openid.php - externí knihovna

Příloha B

Plakát